

38th European Conference on Object-Oriented Programming

ECOOP 2024, September 16–20, 2024, Vienna, Austria

Edited by

Jonathan Aldrich
Guido Salvaneschi



Editors

Jonathan Aldrich 

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

Guido Salvaneschi 

University of St. Gallen, Switzerland
guido.salvaneschi@unisg.ch

ACM Classification 2012

Software and its engineering → General programming languages

ISBN 978-3-95977-341-6

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/978-3-95977-341-6>.

Publication date

September, 2024

Bibliographic information published by the Deutsche Nationalbibliothek

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

License

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



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

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

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.ECOOP.2024.0

ISBN 978-3-95977-341-6

ISSN 1868-8969

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

LIPICS – Leibniz International Proceedings in Informatics

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

Editorial Board

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

ISSN 1868-8969

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

Contents

Message from the Program Chairs <i>Jonathan Aldrich and Guido Salvaneschi</i>	0:ix
Message from the Artifact Evaluation Chairs <i>Karine Even-Mendoza and Raphaël Monat</i>	0:xi
Message from the AITO President <i>Davide Ancona</i>	0:xiii
List of Authors	0:xv

Regular Papers

A Sound Type System for Secure Currency Flow <i>Luca Aceto, Daniele Gorla, and Stian Lybeck</i>	1:1–1:27
Runtime Instrumentation for Reactive Components <i>Luca Aceto, Duncan Paul Attard, Adrian Francalanza, and Anna Ingólfssdóttir</i>	2:1–2:33
A Dynamic Logic for Symbolic Execution for the Smart Contract Programming Language Michelson <i>Barnabas Arvay, Thi Thu Ha Doan, and Peter Thiemann</i>	3:1–3:26
Dynamically Generating Callback Summaries for Enhancing Static Analysis <i>Steven Arzt, Marc Miltenberger, and Julius Näumann</i>	4:1–4:27
Behavioural Up/down Casting For Statically Typed Languages <i>Lorenzo Bacchiani, Mario Bravetti, Marco Giunti, João Mota, and António Ravara</i>	5:1–5:28
Cross Module Quickening – The Curious Case of C Extensions <i>Felix Berlakovich and Stefan Brunthaler</i>	6:1–6:29
HOBBIT: Hashed OBject Based InTegrity <i>Matthias Bernad and Stefan Brunthaler</i>	7:1–7:25
Understanding Concurrency Bugs in Real-World Programs with Kotlin Coroutines <i>Bob Brockbernd, Nikita Koval, Arie van Deursen, and Burcu Kulahcioglu Ozkan</i> ..	8:1–8:20
A Language-Based Version Control System for Python <i>Luis Carvalho and João Costa Seco</i>	9:1–9:27
Indirection-Bounded Call Graph Analysis <i>Madhurima Chakraborty, Aakash Gnanakumar, Manu Sridharan, and Anders Møller</i>	10:1–10:22
Regrading Policies for Flexible Information Flow Control in Session-Typed Concurrency <i>Farzaneh Derakhshan, Stephanie Balzer, and Yue Yao</i>	11:1–11:29

Mutation-Based Lifted Repair of Software Product Lines <i>Aleksandar S. Dimovski</i>	12:1–12:24
Pure Methods for roDOT <i>Vlastimil Dort, Yufeng Li, Ondřej Lhoták, and Pavel Parízek</i>	13:1–13:29
The Performance Effects of Virtual-Machine Instruction Pointer Updates <i>M. Anton Ertl and Bernd Paysan</i>	14:1–14:26
Rose: Composable Autodiff for the Interactive Web <i>Sam Estep, Wode Ni, Raven Rothkopf, and Joshua Sunshine</i>	15:1–15:27
Mover Logic: A Concurrent Program Logic for Reduction and Rely-Guarantee Reasoning <i>Cormac Flanagan and Stephen N. Freund</i>	16:1–16:29
Fair Join Pattern Matching for Actors <i>Philipp Haller, Ayman Hussein, Hernán Melgratti, Alceste Scalas, and Emilio Tuosto</i>	17:1–17:28
A CFL-Reachability Formulation of Callsite-Sensitive Pointer Analysis with Built-In On-The-Fly Call Graph Construction <i>Dongjie He, Jingbo Lu, and Jingling Xue</i>	18:1–18:29
Fearless Asynchronous Communications with Timed Multiparty Session Protocols <i>Ping Hou, Nicolas Lagaillardie, and Nobuko Yoshida</i>	19:1–19:30
Taking a Closer Look: An Outlier-Driven Approach to Compilation-Time Optimization <i>Florian Huemer, David Leopoldseder, Aleksandar Prokopec, Raphael Mosaner, and Hanspeter Mössenböck</i>	20:1–20:28
Learning Gradual Typing Performance <i>Mohammad Wahiduzzaman Khan, Sheng Chen, and Yi He</i>	21:1–21:27
CONSTRICtor: Immutability as a Design Concept <i>Elad Kinsbruner, Shachar Itzhaky, and Hila Peleg</i>	22:1–22:29
InferType: A Compiler Toolkit for Implementing Efficient Constraint-Based Type Inference <i>Senxi Li, Tetsuro Yamazaki, and Shigeru Chiba</i>	23:1–23:28
Qafny: A Quantum-Program Verifier <i>Liyi Li, Mingwei Zhu, Rance Cleaveland, Alexander Nicollellis, Yi Lee, Le Chang, and Xiaodi Wu</i>	24:1–24:31
Compositional Symbolic Execution for Correctness and Incorrectness Reasoning <i>Andreas Lööw, Daniele Nantes-Sobrinho, Sacha-Élie Ayoun, Caroline Cronjäger, Petar Maksimović, and Philippa Gardner</i>	25:1–25:28
Matching Plans for Frame Inference in Compositional Reasoning <i>Andreas Lööw, Daniele Nantes-Sobrinho, Sacha-Élie Ayoun, Petar Maksimović, and Philippa Gardner</i>	26:1–26:20
The Fault in Our Stars: Designing Reproducible Large-scale Code Analysis Experiments <i>Petr Maj, Stefanie Muroya, Konrad Siek, Luca Di Grazia, and Jan Vitek</i>	27:1–27:23

Static Basic Block Versioning <i>Olivier Melançon, Marc Feeley, and Manuel Serrano</i>	28:1–28:27
Generalizing Shape Analysis with Gradual Types <i>Zeina Migeed, James Reed, Jason Ansel, and Jens Palsberg</i>	29:1–29:28
Verifying Lock-Free Search Structure Templates <i>Nisarg Patel, Dennis Shasha, and Thomas Wies</i>	30:1–30:28
Ozone: Fully Out-of-Order Choreographies <i>Dan Plyukhin, Marco Peressotti, and Fabrizio Montesi</i>	31:1–31:28
Tenspiler: A Verified-Lifting-Based Compiler for Tensor Operations <i>Jie Qiu, Colin Cai, Sahil Bhatia, Niranjan Hasabnis, Sanjit A. Seshia, and Alvin Cheung</i>	32:1–32:28
Compiling with Arrays <i>David Richter, Timon Böhler, Pascal Weisenburger, and Mira Mezini</i>	33:1–33:24
Pipit on the Post: Proving Pre- and Post-Conditions of Reactive Systems <i>Amos Robinson and Alex Potanin</i>	34:1–34:28
Partial Redundancy Elimination in Two Iterative Data Flow Analyses <i>Reshma Roy, Sreekala S, and Vineeth Paleri</i>	35:1–35:19
Scaling Interprocedural Static Data-Flow Analysis to Large C/C++ Applications: An Experience Report <i>Fabian Schiebel, Florian Sattler, Philipp Dominik Schubert, Sven Apel, and Eric Bodden</i>	36:1–36:28
Java Bytecode Normalization for Code Similarity Analysis <i>Stefan Schott, Serena Elisa Ponta, Wolfram Fischer, Jonas Klauke, and Eric Bodden</i>	37:1–37:29
Optimizing Layout of Recursive Datatypes with Marmoset: Or, Algorithms + Data Layouts = Efficient Programs <i>Vidush Singhal, Chaitanya Koparkar, Joseph Zullo, Artem Pelenitsyn, Michael Vollmer, Mike Rainey, Ryan Newton, and Milind Kulkarni</i>	38:1–38:28
Formalizing, Mechanizing, and Verifying Class-Based Refinement Types <i>Ke Sun, Di Wang, Sheng Chen, Meng Wang, and Dan Hao</i>	39:1–39:30
Information Flow Control in Cyclic Process Networks <i>Bas van den Heuvel, Farzaneh Derakhshan, and Stephanie Balzer</i>	40:1–40:30
Refinements for Multiparty Message-Passing Protocols: Specification-Agnostic Theory and Implementation <i>Martin Vassor and Nobuko Yoshida</i>	41:1–41:29
Failure Transparency in Stateful Dataflow Systems <i>Aleksey Veresov, Jonas Spenger, Paris Carbone, and Philipp Haller</i>	42:1–42:31
Inductive Predicate Synthesis Modulo Programs <i>Scott Wesley, Maria Christakis, Jorge A. Navas, Richard Trefler, Valentin Wüstholtz, and Arie Gurfinkel</i>	43:1–43:30

Type Tailoring Ashton Wiersdorf, Stephen Chang, Matthias Felleisen, and Ben Greenman	44:1–44:27
Higher-Order Specifications for Deductive Synthesis of Programs with Pointers David Young, Ziyi Yang, Ilya Sergey, and Alex Potanin	45:1–45:26
CtChecker: A Precise, Sound and Efficient Static Analysis for Constant-Time Programming Quan Zhou, Sixuan Dang, and Danfeng Zhang	46:1–46:26
Defining Name Accessibility Using Scope Graphs Aron Zwaan and Casper Bach Poulsen	47:1–47:29

■ Message from the Program Chairs

Started in 1987, ECOOP is Europe's oldest programming conference, welcoming papers on all practical and theoretical investigations of programming languages, systems, and environments that provide innovative solutions to real problems as well as evaluations of existing solutions. Papers were submitted to one of four categories: *Research* for papers that advance the state of the art in programming; *Replication* for empirical evaluations that reconstruct a published experiment in a different context in order to validate the results of that earlier work; *Experience* for applications of known techniques in practice; and *Pearl/Brave New Idea* for papers that either explain a known idea in an elegant way or unconventional papers introducing ideas that may take some time to substantiate. The chairs thank the Program Committee for their dedication to ensuring a quality program and providing constructive feedback to authors: Alvin Cheung, Eva Darulova, Jenna DiVincenzo (Wise), Werner Dietl, Jens Dietrich, Sebastian Erdweg, Patrick Eugster, Carla Ferreira, Simon J. Gay, Jeremy Gibbons, Elisa Gonzalez Boix, Arjun Guha, Suresh Jagannathan, Ranjit Jhala, Yu David Liu, Mira Mezini, Heather Miller, Ragnar Mogk, David Naumann, Marianna Rapoport, António Ravara, Manuel Serrano, Peter Thiemann, Emilio Tuosto, and Elena Zucca.

This year, we continued a number of innovations that were first introduced in 2022:

- **Multiple rounds.** ECOOP has two main rounds of submissions per year (Jan 17 and Apr 17). Each round supports both minor and major revisions. Major revisions are handled in the next round (either the same year or the next) by the same reviewers. In the second round we gave as many papers as possible the chance to try revising by the minor revision deadline so that they could make the 2024 program; all of these resubmissions were accepted.
- **No format or length restrictions.** In order reduce friction for authors, papers can come in any format and at any length. This applies to submissions. Final versions must abide by the publisher's requirements.
- **Artifacts and Papers together.** Every submitted paper can be accompanied with an artifact, submitted a few days after the paper. Both submissions are evaluated in parallel by overlapping committees as members of the artifact evaluation committee were invited to serve on the conference review committee.
- **Journal First/Last.** Papers can be submitted either one of three associated journals and be invited to present at the meeting. Furthermore, some accepted papers can be forwarded to journals.

In addition, this year we introduced a new review process, in which all papers were rated on each of the following criteria:

- **Soundness:** How well the paper's contributions are supported by rigorous application of appropriate research methods;
- **Significance:** The extent to which the paper's contributions are novel, original, and important, with respect to the existing body of knowledge;
- **Presentation:** Whether the paper's quality of writing meets the high standards of ECOOP.

0:x Message from the Program Chairs

After the author response and reviewer discussions, papers were accepted if the PC decided that the paper meets our high bar for Soundness and Presentation, and if at least one reviewer judges the paper to meet the bar for Significance. The goal of this process is to ensure quality of writing and confidence in results, while assuming that if one reviewer finds the paper to be significant then there will be readers who do so as well.

Overall, we found that most of these innovations to have worked well. The new reviewing criteria helped focus the reviewer discussion on what are the main issues with each paper. The acceptance criteria did not affect many papers but made a difference for a few; we believe the result is a more diverse program than might have been accepted based on a traditional, one-dimensional quality rating.

Overall, 59 papers were submitted in the first round and 53 in the second round. Each of these included some resubmissions of papers that received a reject or major revision judgment in prior reviewing rounds. In the end 47 papers were accepted, in many cases after a final round of checking for papers that initially received a conditional accept rating. As is common with journals, the ability to resubmit improved versions of a paper allowed the conference to accept a larger percentage of papers overall than in prior editions of ECOOP, while maintaining a high quality threshold.

We hope that future chairs will continue to experiment with more, and perhaps, different innovations that will enrich the ECOOP community further.

Jonathan Aldrich

Program Committee Co-chair

Carnegie Mellon University

Guido Salvaneschi

Program Committee Co-chair

University of St. Gallen

■ Message from the Artifact Evaluation Chairs

ECOOP has a long-standing tradition of offering artifact evaluation dating back to 2013. Following the process introduced in 2022, the artifact evaluation involved every single paper submission to ECOOP 2024, rather than just accepted papers. As such, it happened in parallel with the paper review process. This approach has two benefits: all authors who submitted an artifact received feedback (independently from paper acceptance), and evaluation results were made available to the reviewers of the papers. In addition, senior artifact evaluation committee members (representing half of the members) contributed to an average of 2 paper reviews to the technical research track as members of the extended review committee, improving the information sharing between the two processes. Artifact submissions could, thus, provide more insights into the technical contributions described in the papers and help to improve the overall review process.

To handle the high review load that such a process entails, we recruited a large artifact evaluation committee that included a total of 61 artifact reviewers. The artifact submissions were due around one week after the paper deadline, for both submission rounds of ECOOP. We received a total of 64 submissions (41 for R1 and 23 for R2). After a kick-the-tires review and author response phase, during which authors had the opportunity to clarify or address technical issues with their submissions, each submitted artifact was reviewed by three committee members.

We have followed ACM's badging policy¹ since 2023; details about the evaluation process are provided in the preface to the Artifact volume.² Out of the 64 submissions, the artifact evaluation committee awarded the highest qualification (available, functional and reusable badges) to 19 artifacts, the available and functional badges to 18 artifacts, and the available badge to 23 artifacts. Out of those 64 submissions, 33 were associated with papers accepted for presentation at ECOOP 2024.

The smooth and thorough artifact evaluation process would have not been possible without the members of the committee, who handled the artifact review workload and contributed to the technical PC discussions with great dedication. We would like to thank them for their valuable work, feedback to authors and the inspiring discussions! We would also like to thank the ECOOP 2024 program committee chairs Guido Salvaneschi and Jonathan Aldrich for the pleasant and productive interactions over the coordination of the paper and artifact review processes, and the Dagstuhl Publishing team for their proactive and highly responsive assistance during the preparation of this DARTS volume.

Karine Even-Mendoza

King's College London

Raphaël Monat

Inria Lille & University of Lille

Artifact Evaluation Co-chairs

¹ <https://www.acm.org/publications/policies/artifact-review-and-badging-current>

² <https://doi.org/10.4230/DARTS.10.2.0>

■ Foreword by the President of AITO

After last year's event in Seattle, ECOOP 2024 returns to the heart of Europe, hosted by the prestigious Vienna University of Technology (TU Wien). Before this year, the conference had been held in Austria only once, 28 years ago in Linz. Therefore, I am especially pleased to welcome the ECOOP community to Vienna.

Although I have been involved in many ECOOP conferences, ECOOP 2024 holds special significance for me, as this is my first time attending the conference as President of AITO. Recently, the AITO Executive Board has undergone significant renewal. I am glad to welcome Christian Hammer and Ben Hermann as the new Secretary and Treasurer of the Board, respectively.

I owe deep gratitude to Eric Jul and Walter Olthoff, our previous President and Treasurer, for supporting us during this transition and for their long-standing contributions to the AITO community and the success of ECOOP.

As in previous years, ECOOP 2024 is co-located with ISSTA and, for the first time, with MPLR. The ECOOP 2024 team, along with the ISSTA and MPLR teams, has done a great job in putting together an excellent and rich program for the conferences, including ten workshops, a doctoral symposium, tool demos, and tutorials. A huge thanks to them and to all others who have contributed.

I am looking forward to excellent conferences and workshops, great presentations fostering personal interaction, and excellent keynotes, including talks by the two 2024 Dahl-Nygaard Prize Winners. Enjoy the conference and Vienna.

Davide Ancona
AITO President

■ List of Authors

- Luca Aceto  (1, 2)
Reykjavík University, Iceland;
Gran Sasso Science Institute, L'Aquila, Italy
- Jason Ansel  (29)
Meta, Menlo Park, CA, USA
- Sven Apel  (36)
Saarland University, Saarland Informatics
Campus, Saarbrücken, Germany
- Barnabas Arvay  (3)
University of Freiburg, Germany
- Steven Arzt  (4)
Fraunhofer SIT | ATHENE – National Research
Center for Applied Cybersecurity, Darmstadt,
Germany
- Duncan Paul Attard  (2)
University of Glasgow, UK
- Sacha-Élie Ayoun (25, 26)
Imperial College London, UK
- Lorenzo Bacchiani  (5)
University of Bologna, Italy
- Casper Bach Poulsen  (47)
Delft University of Technology, The Netherlands
- Stephanie Balzer  (11, 40)
Carnegie Mellon University,
Pittsburgh, PA, USA
- Felix Berlakovich  (6)
University of the Bundeswehr Munich,
Neubiberg, Germany
- Matthias Bernad  (7)
 μ CSRL – Munich Computer Systems Research
Lab, Research Institute CODE, University of
the Bundeswehr Munich, Neubiberg, Germany
- Sahil Bhatia (32)
University of California, Berkeley, CA, USA
- Eric Bodden  (36, 37)
Paderborn University, Department of Computer
Science, Heinz Nixdorf Institute, Germany;
Fraunhofer IEM, Paderborn, Germany
- Mario Bravetti  (5)
University of Bologna, Italy
- Bob Brockbernd (8)
Delft University of Technology, The Netherlands
- Stefan Brunthaler  (6, 7)
University of the Bundeswehr Munich,
Neubiberg, Germany
- Timon Böhler  (33)
Technische Universität Darmstadt, Germany
- Colin Cai (32)
University of California, Berkeley, CA, USA
- Paris Carbone  (42)
EECS and Digital Futures, KTH Royal Institute
of Technology, Stockholm, Sweden;
Digital Systems, RISE Research Institutes of
Sweden, Stockholm, Sweden
- Luís Carvalho  (9)
NOVA LINCS, NOVA School of Science and
Technology, Caparica, Portugal
- Madhurima Chakraborty (10)
University of California, Riverside, CA, USA
- Le Chang (24)
University of Maryland, College Park, MD, USA
- Stephen Chang  (44)
University of Massachusetts Boston, MA, USA
- Sheng Chen  (21, 39)
CACS, University of Louisiana,
Lafayette, LA, USA
- Alvin Cheung (32)
University of California, Berkeley, CA, USA
- Shigeru Chiba  (23)
The University of Tokyo, Japan
- Maria Christakis  (43)
TU Wien, Austria
- Rance Cleaveland (24)
University of Maryland, College Park, MD, USA
- João Costa Seco  (9)
NOVA LINCS, NOVA School of Science and
Technology, Caparica, Portugal
- Caroline Cronjäger (25)
Ruhr-Universität Bochum, Germany
- Sixuan Dang  (46)
Duke University, Durham, NC, USA

- Farzaneh Derakhshan  (11, 40)
Illinois Institute of Technology,
Chicago, IL, USA
- Luca Di Grazia  (27)
Università della Svizzera italiana (USI),
Lugano, Switzerland
- Aleksandar S. Dimovski  (12)
Mother Teresa University,
Skopje, North Macedonia
- Thi Thu Ha Doan  (3)
University of Freiburg, Germany
- Vlastimil Dort  (13)
Charles University, Prague, Czech Republic
- M. Anton Ertl  (14)
TU Wien, Austria
- Sam Estep  (15)
Software and Societal Systems Department,
Carnegie Mellon University, Pittsburgh, PA,
USA
- Marc Feeley  (28)
Université de Montréal, Canada
- Matthias Felleisen  (44)
Northeastern University, Boston, MA, USA
- Wolfram Fischer  (37)
SAP Security Research, Mougins, France
- Cormac Flanagan  (16)
University of California, Santa Cruz, CA, USA
- Adrian Francalanza  (2)
University of Malta, Msida, Malta
- Stephen N. Freund  (16)
Williams College, Williamstown, MA, USA
- Philippa Gardner (25, 26)
Imperial College London, UK
- Marco Giunti  (5)
University of Oxford, UK
- Aakash Gnanakumar (10)
University of California, Riverside, CA, USA
- Daniele Gorla  (1)
Sapienza, Università di Roma, Italy
- Ben Greenman  (44)
University of Utah, Salt Lake City, UT, USA
- Arie Gurfinkel  (43)
University of Waterloo, Canada
- Philipp Haller  (17, 42)
KTH Royal Institute of Technology,
Stockholm, Sweden
- Dan Hao  (39)
Key Lab of HCST (PKU), MOE, School of
Computer Science, Peking University, Beijing,
China
- Niranjan Hasabnis (32)
Intel Labs, Menlo Park, CA, USA
- Dongjie He  (18)
University of New South Wales, Sydney,
Australia; Chongqing University, China
- Yi He  (21)
Data Science, College William & Mary,
Williamsburg, VA, USA
- Ping Hou  (19)
University of Oxford, UK
- Florian Huemer  (20)
Johannes Kepler University, Linz, Austria
- Ayman Hussein  (17)
Technical University of Denmark,
Lyngby, Denmark
- Anna Ingólfssdóttir  (2)
Reykjavík University, Iceland
- Shachar Itzhaky  (22)
Technion, Haifa, Israel
- Mohammad Wahiduzzaman Khan  (21)
CACS, University of Louisiana,
Lafayette, LA, USA
- Elad Kinsbruner  (22)
Technion, Haifa, Israel
- Jonas Klauke  (37)
Paderborn University, Germany
- Chaitanya Koparkar  (38)
Indiana University, Bloomington, IN, USA
- Nikita Koval (8)
JetBrains, Amsterdam, The Netherlands
- Milind Kulkarni  (38)
Purdue University, West Lafayette, IN, USA
- Nicolas Lagaillardie  (19)
Imperial College London, UK
- Yi Lee (24)
University of Maryland, College Park, MD, USA

- David Leopoldseder  (20)
Oracle Labs, Vienna, Austria
- Ondřej Lhoták  (13)
University of Waterloo, Canada
- Liyi Li  (24)
Iowa State University, Ames, IA, USA
- Senxi Li  (23)
The University of Tokyo, Japan
- Yufeng Li (13)
University of Cambridge, UK
- Jingbo Lu  (18)
University of New South Wales, Sydney, Australia; Shanghai Sectrend Information Technology Co., Ltd, China
- Stian Lybeck  (1)
Reykjavík University, Iceland
- Andreas Lööw (25, 26)
Imperial College London, UK
- Petr Maj (27)
Czech Technical University, Prague, Czech Republic
- Petar Maksimović (25, 26)
Imperial College London, UK; Runtime Verification Inc., Chicago, IL, USA
- Olivier Melançon  (28)
Université de Montréal, Canada
- Hernán Melgratti  (17)
University of Buenos Aires & Conicet, Argentina
- Mira Mezini  (33)
Technische Universität Darmstadt, Germany; The Hessian Center for Artificial Intelligence (hessian.AI), Darmstadt, Germany
- Zeina Migeed  (29)
University of California, Los Angeles (UCLA), CA, USA
- Marc Miltenberger  (4)
Fraunhofer SIT | ATHENE – National Research Center for Applied Cybersecurity, Darmstadt, Germany
- Fabrizio Montesi  (31)
University of Southern Denmark, Odense, Denmark
- Raphael Mosaner  (20)
Oracle Labs, Linz, Austria
- João Mota  (5)
NOVA LINCS, Nova University Lisbon, Portugal;
NOVA School of Science and Technology, Caparica, Portugal
- Stefanie Muroya (27)
Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria
- Hanspeter Mössenböck  (20)
Johannes Kepler University, Linz, Austria
- Anders Møller  (10)
Aarhus University, Denmark
- Daniele Nantes-Sobrinho (25, 26)
Imperial College London, UK
- Jorge A. Navas  (43)
Certora, Seattle, WA, USA
- Ryan Newton  (38)
Purdue University, West Lafayette, IN, USA
- Wode Ni  (15)
Software and Societal Systems Department, Carnegie Mellon University, Pittsburgh, PA, USA
- Alexander Nicolellis (24)
Iowa State University, Ames, IA, USA
- Julius Näumann  (4)
TU Darmstadt | ATHENE – National Research Center for Applied Cybersecurity, Darmstadt, Germany
- Burcu Kulahcioglu Ozkan  (8)
Delft University of Technology, The Netherlands
- Vineeth Paleri  (35)
National Institute of Technology Calicut, India
- Jens Palsberg  (29)
University of California, Los Angeles (UCLA), CA, USA
- Pavel Parízek  (13)
Charles University, Prague, Czech Republic
- Nisarg Patel  (30)
New York University, NY, USA
- Bernd Paysan (14)
net2o, Munich, Germany
- Hila Peleg  (22)
Technion, Haifa, Israel
- Artem Pelenitsyn  (38)
Purdue University, West Lafayette, IN, USA

- Marco Peressotti  (31)
University of Southern Denmark,
Odense, Denmark
- Dan Plyukhin  (31)
University of Southern Denmark,
Odense, Denmark
- Serena Elisa Ponta  (37)
SAP Security Research, Mougins, France
- Alex Potanin  (34, 45)
Australian National University,
Canberra, Australia
- Aleksandar Prokopec  (20)
Oracle Labs, Zurich, Switzerland
- Jie Qiu  (32)
Pittsburgh, PA, USA
- Mike Rainey  (38)
Carnegie Mellon University,
Pittsburgh, PA, USA
- António Ravara  (5)
NOVA LINCS, Nova University Lisbon,
Portugal; NOVA School of Science and
Technology, Caparica, Portugal
- James Reed (29)
Fireworks AI, Redwood City, CA, USA
- David Richter  (33)
Technische Universität Darmstadt, Germany
- Amos Robinson  (34)
Sydney, Australia
- Raven Rothkopf  (15)
Barnard College, Columbia University,
New York, NY, USA
- Reshma Roy  (35)
National Institute of Technology, Calicut, India
- Sreekala S  (35)
National Institute of Technology Calicut, India
- Florian Sattler  (36)
Saarland University, Saarland Informatics
Campus, Saarbrücken, Germany
- Alceste Scalas  (17)
Technical University of Denmark,
Lyngby, Denmark
- Fabian Schiebel  (36)
Fraunhofer Institute for Mechatronic Systems
Design IEM, Paderborn, Germany
- Stefan Schott  (37)
Paderborn University, Germany
- Philipp Dominik Schubert  (36)
Heinz Nixdorf Institute, Paderborn, Germany
- Ilya Sergey  (45)
National University of Singapore, Singapore
- Manuel Serrano  (28)
Inria/UCA, Inria Sophia Méditerranée,
Sophia Antipolis, France
- Sanjit A. Seshia (32)
University of California, Berkeley, CA, USA
- Dennis Shasha  (30)
New York University, NY, USA
- Konrad Siek (27)
Czech Technical University,
Prague, Czech Republic
- Vidush Singhal  (38)
Purdue University, West Lafayette, IN, USA
- Jonas Spenger  (42)
EECS and Digital Futures, KTH Royal Institute
of Technology, Stockholm, Sweden
- Manu Sridharan  (10)
University of California, Riverside, CA, USA
- Ke Sun  (39)
Key Lab of HCST (PKU), MOE, School of
Computer Science, Peking University, Beijing,
China
- Joshua Sunshine  (15)
Software and Societal Systems Department,
Carnegie Mellon University, Pittsburgh, PA,
USA
- Peter Thiemann  (3)
University of Freiburg, Germany
- Richard Trefler  (43)
University of Waterloo, Canada
- Emilio Tuosto  (17)
Gran Sasso Science Institute, L'Aquila, Italy
- Bas van den Heuvel  (40)
HKA Karlsruhe, Germany; University of
Freiburg, Germany; University of Groningen,
The Netherlands
- Arie van Deursen  (8)
Delft University of Technology, The Netherlands
- Martin Vassor  (41)
University of Oxford, UK
- Aleksey Veresov  (42)
EECS and Digital Futures, KTH Royal Institute
of Technology, Stockholm, Sweden

- Jan Vitek (27)
Charles University, Prague, Czech Republic;
Northeastern University, Boston, MA, USA
- Michael Vollmer  (38)
University of Kent, UK
- Di Wang  (39)
Key Lab of HCST (PKU), MOE, School of
Computer Science, Peking University, Beijing,
China
- Meng Wang  (39)
University of Bristol, UK
- Pascal Weisenburger  (33)
University of St. Gallen, Switzerland
- Scott Wesley  (43)
Dalhousie University, Halifax, Canada
- Ashton Wiersdorf  (44)
University of Utah, Salt Lake City,
UT, USA
- Thomas Wies  (30)
New York University, NY, USA
- Xiaodi Wu  (24)
University of Maryland,
College Park, MD, USA
- Valentin Wüstholtz (43)
ConsenSys, Vienna, Austria
- Jingling Xue  (18)
University of New South Wales,
Sydney, Australia
- Tetsuro Yamazaki  (23)
The University of Tokyo, Japan
- Ziyi Yang  (45)
National University of Singapore, Singapore
- Yue Yao  (11)
Carnegie Mellon University,
Pittsburgh, PA, USA
- Nobuko Yoshida  (19, 41)
University of Oxford, UK
- David Young  (45)
University of Kansas, Lawrence, KS, USA
- Danfeng Zhang  (46)
Duke University, Durham, NC, USA
- Quan Zhou  (46)
Penn State University,
University Park, PA, USA

- Mingwei Zhu (24)
University of Maryland, College Park, MD, USA
- Joseph Zullo  (38)
Purdue University, West Lafayette, IN, USA
- Aron Zwaan  (47)
Delft University of Technology, The Netherlands

A Sound Type System for Secure Currency Flow

Luca Aceto 

Reykjavík University, Iceland

Daniele Gorla 

Sapienza, Università di Roma, Italy

Stian Lybech 

Reykjavík University, Iceland

Abstract

In this paper we focus on TINY SOL, a minimal calculus for Solidity smart contracts, introduced by Bartoletti et al. We start by rephrasing its syntax (to emphasise its object-oriented flavour) and give a new big-step operational semantics. We then use it to define two security properties, namely call integrity and noninterference. These two properties have some similarities in their definition, in that they both require that some part of a program is not influenced by the other part. However, we show that the two properties are actually incomparable. Nevertheless, we provide a type system for noninterference and show that well-typed programs satisfy call integrity as well; hence, programs that are accepted by our type system satisfy both properties. We finally discuss the practical usability of the type system and its limitations by means of some simple examples.

2012 ACM Subject Classification Theory of computation → Program analysis; Theory of computation → Type structures

Keywords and phrases smart contracts, call integrity, noninterference, type system

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.1

Related Version Full Version: <https://arxiv.org/abs/2405.12976> [1]

Funding Luca Aceto: Supported by the Icelandic Research Fund Grant No. 218202-05(1-3).

Stian Lybech: Supported by the Icelandic Research Fund Grant No. 218202-05(1-3).

Acknowledgements We thank the anonymous reviewers for their constructive attitude and for the fruitful comments that helped us improve our paper. Luca Aceto and Stian Lybech thank Mohammad Hamdaqa for sharing his expertise with them during extensive discussions on safety properties for smart contracts, which helped shape the research agenda for the work reported in this paper.

1 Introduction

The classic notion of noninterference [12] is a well-known concept that has been applied in a variety of settings to characterise both integrity and secrecy in programming. In particular, this property has been defined by Volpano et al. [28] in terms of a lattice model of security levels (e.g. “High” and “Low”, or “Trusted” and “Untrusted”); the key point being that information must not flow from a higher to a lower level. Thus, the lower levels are unaffected by the higher ones, and, conversely, the higher levels are “noninterfering” with the lower ones.

Ensuring noninterference seems particularly relevant in a setting where not only information, but also *currency*, flows between programs. This is a core feature of *smart contracts*, which are programs that run atop a blockchain and are used to manage financial assets of users, codify transactions, and implement custom tokens; see e.g. [24] for an overview of the architecture. The code of a smart contract resides on the blockchain itself, and is therefore both immutable and publicly visible. This is one of the important ways in which the “smart-contract programming paradigm” differs from conventional programming languages.



© Luca Aceto, Daniele Gorla, and Stian Lybech;

licensed under Creative Commons License CC-BY 4.0

38th European Conference on Object-Oriented Programming (ECOOP 2024).

Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 1; pp. 1:1–1:27



Leibniz International Proceedings in Informatics

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

1:2 A Sound Type System for Secure Currency Flow

```

1  contract X {
2    ...
3    field called := F;
4    transfer(z) {
5      if  $\neg$ called  $\wedge$  this.balance  $\geq$  1
6        then z.deposit(this):1;
7          this.called := T;
8        else skip
9    }
10 }
```



```

contract Y {
  ...
  deposit(x) {
    x.transfer(this):0
  }
}
```

Figure 1 Illustration of reentrancy written in the language TINY SOL.

Public visibility means that *vulnerabilities* in the code can be found and exploited by a malicious user. Moreover, if a vulnerability is discovered, immutability prevents the contract creator from correcting the error. Thus, it is obviously desirable to ensure that a smart contract is safe and correct *before* it is deployed onto the blockchain.

The combination of immutability and visibility has led to huge financial losses in the past (see, e.g., [2, 8, 19, 20, 26]). A particularly spectacular example was the infamous DAO-attack on the Ethereum platform in 2016, which led to a loss of 60 million dollars [8]. This was made possible because a certain contract (the DAO contract, storing assets of users) was *reentrant*, that is, it allowed itself to be called back by the recipient of a transfer *before* recording that the transfer had been completed.

Reentrancy is a pattern based on mutual recursion, where one method f calls another method g whilst also transferring an amount of currency along with the call. If g then immediately calls f back, it may yield a recursion where f will keep transferring funds to g . We can illustrate the problem as in Figure 1, using a simple, imperative and class-based model language called TINY SOL [3]. This model language, which we shall formally describe in Section 2, captures some of the core features of the smart-contract language Solidity [10], which is the standard high-level language used to write smart contracts for the Ethereum platform. A key feature of this language is that contracts have an associated **balance**, representing the amount of currency stored in each contract, which cannot be modified *except* through method calls to other contracts. Each method call has an extra parameter, representing the amount of currency to be transferred along with the call, and a method call thus represents a (potential) outgoing currency flow.

In Figure 1, **X.transfer(z)** first does a sanity check to ensure that it has not already been called and that the contract contains sufficient funds, which are stored in the **balance** field. Then it calls **z.deposit(this)** and transfers 1 unit of currency along with the call, where **z** is the address received as parameter. However, suppose the address received is **Y**. Then **Y.deposit(x)** immediately calls **X.transfer(z)** back, with **this** as actual parameter; this yields a mutual recursion, because the field **called** will never be set to **T**. A transaction that invokes **X.transfer(Y)** with any number of currency units will trigger the recursion.

The problem is that currency cannot be transferred without also transferring control to the recipient, and the execution of **X.transfer(z)** comes to depend on unknown and untrusted code in the contract residing at the address received as the actual parameter. Simply switching the order of lines 6 and 7 in **X** solves the problem in this particular case, but it might not always be possible to move external calls to the last position in a sequence of statements. Furthermore, the execution of a function f can also depend on external fields, and not only on external calls. Thus, reentrancy is not just a purely syntactic property.

The property of reentrancy in Ethereum smart contracts has been formally characterised by Grishchenko et al. in [13]. Specifically, they define another property, named *call integrity*, which implies the absence of reentrancy (see [13, Theorem 1]) and has been identified in the literature as one of the safety properties that smart contracts should have. Informally, this property requires any call to a method in a “trusted” contract (say, X) to yield the exact same sequence of currency flows (i.e. method calls) even if some of the other “untrusted” contracts (or their stored values) are changed. In a sense, the code and values of the other contracts, which could be controlled by an attacker, must not be able to affect the currency flow from X .

A disadvantage of the definition of call integrity given in [13] is that it relies on a universal quantification over all possible execution contexts, which makes it hard to be checked in practice. However, call integrity seems intuitively to be related to noninterference, in the sense that both stipulate that changes in one part of a program should not have an effect upon another part. Even though we discover that the two properties are incomparable, one might hope to be able to apply techniques for ensuring noninterference to also capture call integrity. Specifically, Volpano et al. [28] show that noninterference can be soundly approximated using a type system. In the present paper, we shall therefore create an adaptation of this type system for secure-flow analysis to the setting of smart contracts and show that the resulting type system *also* captures call integrity.

To recap, our main contributions in this paper are: (1) a thorough study of the connections between call integrity and noninterference for smart contracts written in the language TINY SOL, and (2) a sound type system guaranteeing (noninterference and) call integrity for programs written in that language. We choose TINY SOL because it provides a minimal calculus for Solidity contracts and thus allows us to focus on the gist of our main contributions in a simple setting. In doing this, we also provide a simpler operational semantics for this language; this can be considered a third contribution of our work.

The paper is organised as follows: In Section 2, we describe a revised version of the smart-contract language TINY SOL [3]. In Section 3, we adapt the definition of call integrity from [13] and of noninterference from [25] to this language; we then show that these two desirable properties are actually incomparable. Nevertheless, there *is* an overlap between them. In Section 4, we create a type system for ensuring noninterference in TINY SOL, along the lines of Volpano et al. [28], and prove a type soundness result (Theorems 12–15). Our main result is Theorem 19, which shows that well-typedness provides a sound approximation to *both* noninterference *and* call integrity. This is used on a few examples in Section 5, where we also discuss the limitations of the type system. We survey some related work in Section 6 and conclude the paper with some directions for future research in Section 7. All proofs and some technical details are omitted from this paper for space reasons; they can be found in [1].

2 The TinySol language

In [3], Bartoletti et al. present the TINY SOL language, a standard imperative language (similar to Dijkstra’s `While` language [18]), extended with classes (contracts) and two constructs: (1) a `throw` command, representing a fatal error, and (2) a procedure call, with an extra parameter n , denoting an amount of some digital asset, which is transferred along with the call from the caller to the callee. TINY SOL captures (some of) the core features of Solidity, and, in particular, it is sufficient to represent reentrancy phenomena. In this section, we present a version of TINY SOL which has been adapted to facilitate our later developments of the type system. Compared to the presentation in [3], we have, in particular, added explicit declarations of variables (local to the scope of a method) and fields (corresponding to the *keys* in the original presentation) to have a place for type annotations in the syntax.

$$\begin{aligned}
DF \in \text{Dec}_F &::= \epsilon \mid \text{field } p := v; DF \\
DM \in \text{Dec}_M &::= \epsilon \mid f(\tilde{x}) \{ S \} DM \\
DC \in \text{Dec}_C &::= \epsilon \mid \text{contract } X \{ \\
&\quad \text{field balance} := n; DF \\
&\quad \text{send()} \{ \text{skip} \} DM \\
&\} DC \\
m \in \text{MVar} &::= \text{this} \mid \text{sender} \mid \text{value} \\
L \in \text{LVal} &::= x \mid \text{this}.p \\
e \in \text{Exp} &::= v \mid x \mid m \mid e.\text{balance} \mid e.p \mid \text{op}(\tilde{e}) \\
S \in \text{Stm} &::= \text{skip} \mid \text{throw} \mid \text{var } x := e \text{ in } S \mid L := e \mid S_1; S_2 \\
&\mid \text{if } e \text{ then } S_T \text{ else } S_F \mid \text{while } e \text{ do } S \mid e_1.f(\tilde{e}): e_2 \\
v \in \text{Val} &::= \mathbb{N} \cup \mathbb{B} \cup \text{ANames}
\end{aligned}$$

where $x, y \in \text{VNames}$ (variable names), $p, q \in \text{FNames}$ (field names),
 $X, Y \in \text{ANames}$ (address names), $f, g \in \text{MNames}$ (method names)

■ **Figure 2** The syntax of TINY SOL.

2.1 Syntax

The syntax of TINY SOL is given in Figure 2, where we use the notation $\tilde{\cdot}$ to denote (possibly empty) sequences of items. The set of *values*, ranged over by v , is formed by the sets of integers \mathbb{N} , ranged over by n , booleans $\mathbb{B} = \{\text{T}, \text{F}\}$, ranged over by b , and address names ANames , ranged over by X, Y .

We introduce explicit declarations for fields DF , methods DM , and contracts DC . The latter also encompasses declarations of accounts: an *account* is a contract that contains only the declarations of a special field **balance** and of a single special method **send()**, which does nothing and is used only for transferring funds to the account. By contrast, a contract usually contains other declarations of fields and methods. For the sake of simplicity, we make no syntactic distinction between an account and a contract but, for the purpose of distinguishing, we can assume that the set ANames is split into contract addresses and account addresses.

We have four “magic” keywords in our syntax:

- **balance** (type **int**), a special field recording the current balance of the contract (or account). It can be read from, but not directly assigned to, except through method calls. This ensures that the total amount of currency “on-chain” remains constant during execution.
- **value** (type **int**), a special variable that is bound to the currency amount transferred with a method call.
- **sender** (type **address**), a special variable that is always bound to the address of the caller of a method.
- **this** (type **address**), a special variable that is always bound to the address of the contract containing the currently executing method.

The last three of these are local variables, and we collectively refer to them as “magic variables” $m \in \text{MVar}$. The declaration of variables and fields are very alike: the main difference is that variable bindings will be created at runtime (and with scoped visibility), hence we can let the initial assignment be an *expression* e ; whilst the initial assignment to fields must be *values* v .

The core part of the language is the declaration of expressions e and statements S , that are almost the same as in [3]. The main differences are: (1) we introduce fields p in expressions, instead of keys; (2) we explicitly distinguish between (global) fields and (local) variables, where the latter are declared with a scope limited to a statement S ; and (3) we introduce explicit *lvalues* L , to restrict what can appear on the left-hand side of an assignment (in particular, this ensures that the special field `balance` can never be assigned to directly).

As in the original presentation of TINY SOL, we can also use our new formulation of the language to describe transactions and blockchains. A *transaction* is simply a call, where the caller is an account A , rather than a contract. We denote this by writing $A \rightarrow X.f(\tilde{v}):n$, which expresses that the account A calls the method f on the contract (residing at address) X , with actual parameters \tilde{v} , and transferring n amount of currency with the call. We can then model blockchains as follows:

► **Definition 1** (Syntax of blockchains). *A blockchain $B \in \mathcal{B}$ is a list of initial contract declarations DC , followed by a sequence of transactions $T \in Tr$:*

$$B ::= DC \ T \quad T ::= \epsilon \mid A \rightarrow X.f(\tilde{v}):n, T$$

Notationally, a blockchain with an empty DC will be simply written as the sequence of transactions.

2.2 Big-step semantics

To define the semantics, we need some environments to record the bindings of variables (including the three magic variable names `this`, `sender` and `value`), fields, methods, and contracts. We define them as sets of partial functions as follows:

► **Definition 2** (Binding model). *We define the following sets of partial functions:*

$$\begin{array}{ll} \text{env}_V \in \text{Env}_V : \text{VNames} \cup \text{MVar} \rightarrow \text{Val} & \text{env}_S \in \text{Env}_S : \text{ANames} \rightarrow \text{Env}_F \\ \text{env}_F \in \text{Env}_F : \text{FNames} \cup \{\text{balance}\} \rightarrow \text{Val} & \text{env}_T \in \text{Env}_T : \text{ANames} \rightarrow \text{Env}_M \\ \text{env}_M \in \text{Env}_M : \text{MNames} \rightarrow \text{VNames}^* \times \text{Stm} & \end{array}$$

We regard each environment env_X , for any $X \in \{V, F, M, S, T\}$, as a list of pairs (d, c) where $d \in \text{dom}(\text{env}_X)$ and $c \in \text{codom}(\text{env}_X)$. The notation $\text{env}_X[d \mapsto c]$ denotes the update of env_X mapping d to c . We write env_X^\emptyset for the empty environment. To simplify the notation, when two or more environments appear together, we shall use the convention of writing the subscripts together (e.g. env_{MF} instead of $\text{env}_M, \text{env}_F$).

Our binding model consists of two environments: a *method table* env_T , which maps addresses to method environments, and a *state* env_S , which maps addresses to lists of fields and their values. Thus, for each contract, we have the list of methods it declares and its current state; of course, the method table is constant, once all declarations are performed, whereas the state will change during the evaluation of a program.

2.2.1 Declarations

The semantics of declarations builds the field and method environments, env_F and env_M , and the state and method table env_S and env_T . We give the semantics in a classic big-step style; thus, transitions are of the form $\langle DX, \text{env}_X \rangle \rightarrow_{DX} \text{env}'_X$ for $X \in \{F, M, C, S, T\}$, and their defining rules are given in Figure 3. Notationally, here and in what follows, we denote

$$\begin{array}{c}
[\text{DEC-F}_1] \frac{}{\langle \epsilon, \text{env}_F \rangle \rightarrow_{DF} \text{env}_F} \quad [\text{DEC-F}_2] \frac{\langle DF, \text{env}_F \rangle \rightarrow_{DF} \text{env}'_F}{\langle \text{field } p := v; DF, \text{env}_F \rangle \rightarrow_{DF} (p, v) : \text{env}'_F} \\
[\text{DEC-M}_1] \frac{}{\langle \epsilon, \text{env}_M \rangle \rightarrow_{DM} \text{env}_M} \quad [\text{DEC-M}_2] \frac{\langle DM, \text{env}_M \rangle \rightarrow_{DM} \text{env}'_M}{\langle f(\tilde{x}) \in S \models DM, \text{env}_M \rangle \rightarrow_{DM} (f, (\tilde{x}, S)) : \text{env}'_M} \\
[\text{DEC-C}_1] \frac{}{\langle \epsilon, \text{env}_{ST} \rangle \rightarrow_{DC} \text{env}_{ST}} \\
[\text{DEC-C}_2] \frac{\langle DF, \text{env}_F^\emptyset \rangle \rightarrow_{DF} \text{env}_F \quad \langle DM, \text{env}_M^\emptyset \rangle \rightarrow_{DM} \text{env}_M \quad \langle DC, \text{env}_{ST} \rangle \rightarrow_{DC} \text{env}'_{ST}}{\langle \text{contract } X \{ DF \ DM \} DC, \text{env}_{ST} \rangle \rightarrow_{DC} (X, \text{env}_F) : \text{env}'_S, (X, \text{env}_M) : \text{env}'_T}
\end{array}$$

■ **Figure 3** Semantics of declarations.

$$\begin{array}{c}
[\text{EXP-VAR}] \frac{k \in \text{dom}(\text{env}_V) \quad \text{env}_V(k) = v}{\text{env}_{SV} \vdash k \rightarrow_e v} \quad [\text{EXP-OP}] \frac{\text{env}_{SV} \vdash \tilde{e} \rightarrow_e \tilde{v} \quad \text{op}(\tilde{v}) \rightarrow_{\text{op}} v}{\text{env}_{SV} \vdash \text{op}(\tilde{e}) \rightarrow_e v} \\
[\text{EXP-VAL}] \frac{}{\text{env}_{SV} \vdash v \rightarrow_e v} \quad [\text{EXP-FIELD}] \frac{\text{env}_{SV} \vdash e \rightarrow_e X \quad q \in \text{dom}(\text{env}_S(X)) \quad \text{env}_S(X)(q) = v}{\text{env}_{SV} \vdash e.q \rightarrow_e v}
\end{array}$$

■ **Figure 4** Semantics of expressions.

with $e : l$ the list that results from prepending an element e to the list l . We assume that field and method names are distinct within each contract; therefore, the rules in Figure 3 define partial, finite functions.

2.2.2 Expressions

Figure 4 gives the semantics of expressions e . Expressions have no side effects, so they cannot contain method calls, but they can access both local variables and fields of any contract. Thus expression evaluations are of the form $\text{env}_{SV} \vdash e \rightarrow_e v$, i.e. they are relative to the state and variable environments. We use k to range over `this`, `sender`, `value` and variables x (i.e. $k \in \text{dom}(\text{env}_V)$), and q to range over `balance` and fields p (i.e. $q \in \text{dom}(\text{env}_F)$).

We do not give explicit rules for the boolean and integer operators subsumed under op , but simply assume that they can be evaluated to a unique value by some semantics $\text{op}(\tilde{v}) \rightarrow_{\text{op}} v$.¹ It follows that each expression evaluates to a unique value relative to some given state and variable environments. Note that we assume that no operation is defined for addresses X , so we disallow any form of pointer arithmetic.

2.2.3 Statements

The semantics of statements describes the actual execution steps of a program. In Figure 5 we give the semantics in big-step style, where a step describes the execution of a statement in its entirety. Statements can read from the method table and they can modify the state (i.e., the variable and field bindings). The result of executing a statement is a new state, so transitions must here be of the form $\text{env}_T \vdash \langle S, \text{env}_{SV} \rangle \rightarrow_S \text{env}'_{SV}$ (recall that env'_{SV} stands for $\text{env}'_S, \text{env}'_V$), since both the field values in env_S and the values of the local variables in env_V may have been modified by the execution of S .

¹ To simplify the definitions, we assume that all operations are total. If this was not the case, we would have needed some exception handling for partial operations (e.g., division by zero).

$$\begin{array}{c}
[\text{BS-SKIP}] \frac{}{\text{env}_T \vdash \langle \text{skip}, \text{env}_{SV} \rangle \rightarrow_S \text{env}_{SV}} \\
\\
[\text{BS-SEQ}] \frac{\text{env}_T \vdash \langle S_1, \text{env}_{SV} \rangle \rightarrow_S \text{env}_{SV}'' \quad \text{env}_T \vdash \langle S_2, \text{env}_{SV}'' \rangle \rightarrow_S \text{env}_{SV}'}
{\text{env}_T \vdash \langle S_1 ; S_2, \text{env}_{SV} \rangle \rightarrow_S \text{env}_{SV}'}
\\
\\
[\text{BS-IF}] \frac{\text{env}_{SV} \vdash e \rightarrow_e b \quad \text{env}_T \vdash \langle S_b, \text{env}_{SV} \rangle \rightarrow_S \text{env}_{SV}' \quad (b \in \{\text{T}, \text{F}\})}
{\text{env}_T \vdash \langle \text{if } e \text{ then } S_{\text{T}} \text{ else } S_{\text{F}}, \text{env}_{SV} \rangle \rightarrow_S \text{env}_{SV}'}
\\
\\
[\text{BS-LOOP_T}] \frac{\text{env}_{SV} \vdash e \rightarrow_e \text{T} \quad \text{env}_T \vdash \langle S, \text{env}_{SV} \rangle \rightarrow_S \text{env}_{SV}'' \quad \text{env}_T \vdash \langle \text{while } e \text{ do } S, \text{env}_{SV}'' \rangle \rightarrow_S \text{env}_{SV}'}
{\text{env}_T \vdash \langle \text{while } e \text{ do } S, \text{env}_{SV} \rangle \rightarrow_S \text{env}_{SV}'}
\\
\\
[\text{BS-LOOP_F}] \frac{\text{env}_{SV} \vdash e \rightarrow_e \text{F}}
{\text{env}_T \vdash \langle \text{while } e \text{ do } S, \text{env}_{SV} \rangle \rightarrow_S \text{env}_{SV}}
\\
\\
[\text{BS-DECV}] \frac{x \notin \text{dom}(\text{env}_V) \quad \text{env}_{SV} \vdash e \rightarrow_e v \quad \text{env}_T \vdash \langle S, \text{env}_S, (x, v) : \text{env}_V \rangle \rightarrow_S \text{env}'_S, (x, v') : \text{env}'_V}
{\text{env}_T \vdash \langle \text{var } x := e \text{ in } S, \text{env}_{SV} \rangle \rightarrow_S \text{env}'_{SV}}
\\
\\
[\text{BS-AssV}] \frac{x \in \text{dom}(\text{env}_V) \quad \text{env}_{SV} \vdash e \rightarrow_e v}
{\text{env}_T \vdash \langle x := e, \text{env}_{SV} \rangle \rightarrow_S \text{env}_S, \text{env}_V[x \mapsto v]}
\\
\\
[\text{BS-AssF}] \frac{\text{env}_V(\text{this}) = X \quad \text{env}_S(X) = \text{env}_F \quad p \in \text{dom}(\text{env}_F) \quad \text{env}_{SV} \vdash e \rightarrow_e v}
{\text{env}_T \vdash \langle \text{this}.p := e, \text{env}_{SV} \rangle \rightarrow_S \text{env}_S[X \mapsto \text{env}_F[p \mapsto v]], \text{env}_V}
\\
\\
[\text{BS-CALL}] \frac{\begin{array}{l} \text{env}_{SV} \vdash e_1 \rightarrow_e Y \quad \text{env}_S(Y) = \text{env}_F^Y \quad (\text{env}_T(Y))(f) = (\tilde{x}, S) \\ |\tilde{x}| = |\tilde{e}| = k \quad \text{env}_{SV} \vdash \tilde{e} \rightarrow_e \tilde{v} \quad \text{env}_{SV} \vdash e_2 \rightarrow_e n \\ \text{env}_V(\text{this}) = X \quad \text{env}_S(X) = \text{env}_F^X \quad n \leq \text{env}_F^X(\text{balance}) \\ \text{env}'_S = \text{env}_S[X \mapsto \text{env}_F^X[\text{balance} == n]] \quad [Y \mapsto \text{env}_F^Y[\text{balance} += n]] \\ \text{env}'_V = (\text{this}, Y) : (\text{sender}, X) : (\text{value}, n) : (x_1, v_1) : \dots : (x_k, v_k) : \text{env}_V^\emptyset \end{array}}{\text{env}_T \vdash \langle S, \text{env}'_{SV} \rangle \rightarrow_S \text{env}'_{SV}}
\frac{}{\text{env}_T \vdash \langle e_1.f(\tilde{e}) : e_2, \text{env}_{SV} \rangle \rightarrow_S \text{env}'_S, \text{env}_V}
\end{array}$$

Figure 5 Big-step semantics of statements in TINY SOL.

Most of the rules are straightforward. The rule [BS-DECV] is used when we declare a new variable x , with scope limited to the statement S ; we implicitly assume alpha-conversion to handle shadowing of an existing name. In the premise, we evaluate the expression e to a value v , and then execute the statement S with a variable environment $(x, v) : \text{env}_V$, where we have added the pair (x, v) . During the execution of S , this variable environment may of course be updated (by applications of the rule [BS-AssV]), which may alter any value in the environment, including v . However, outside of the scope of the declaration, x is not visible and so the pair (x, v') is removed from the environment once S finishes. By contrast, any other change made to env'_V (as well as any change made to the global state env_S) is retained.

The [BS-CALL] rule is the most complicated, because we need to perform a number of actions. Some of them are obvious (e.g., evaluate the address and the parameters e_1 , \tilde{e} and e_2 , relatively to the current execution environment env_{SV} ; use the obtained address Y of the callee to retrieve the field environment env_F^Y for this contract and, through the method table, to extract the list of formal parameters \tilde{x} and the body of the method S ; and check that the number of actual parameters is the same as the number of formal parameters). Then, we also have to check that the `balance` of the caller is at least n , and, in that case,

$$\begin{array}{c}
 [\text{GENESIS}] \frac{\langle DC, \text{env}_{ST}^\emptyset \rangle \rightarrow_{DC} \text{env}_{ST}}{\langle DC, T, \text{env}_{ST}^\emptyset \rangle \rightarrow_B \langle T, \text{env}_{ST} \rangle} \qquad [\text{REVELATION}] \frac{}{\langle \epsilon, \text{env}_{ST} \rangle \rightarrow_B \text{env}_{ST}} \\
 \\
 [\text{TRANS}] \frac{\text{env}_T \vdash \langle X.f(\tilde{v}):n, \text{env}_S, (\text{this}, A) : \text{env}_V^\emptyset \rangle \rightarrow_S \text{env}'_S, \text{env}_V}{\langle A \rightarrow X.f(\tilde{v}):n, T, \text{env}_{ST} \rangle \rightarrow_B \langle T, \text{env}'_S, \text{env}_T \rangle}
 \end{array}$$

 **Figure 6** Semantics of blockchains.

update the state environment by subtracting n from the balance of X and adding n to the balance of Y , in their respective field environments; this yields a new state env'_S , where we write $\text{env}_F[\text{balance} == n]$ and $\text{env}'_F[\text{balance} += n]$ for these two operations. Finally, we create the new execution environment by creating new bindings for the special variables `this`, `sender` and `value`, and by binding the formal parameters \tilde{x} to the values of the actual parameters \tilde{v} in env'_V . Then we execute the statement S in this new environment. This yields the new state env'_S , and also an updated variable environment env'_V , since S may have modified the bindings in env'_V . However, these bindings are local to the method, and therefore we throw them away once the call finishes. So, the result of this transition is the updated state env'_S and the original variable environment of the caller env_V .

It should be noted that a *local* method call, i.e. a call to a method within the same (calling) contract, is merely a special case of the rule [BS-CALL]. Such a call would have the form `this.f(̄e):0`, since transferring any amount of currency will not alter the balance of the contract. Thus, we could introduce some syntactic sugar, omitting both the address and the value, and instead simply write $f(\tilde{e})$.

2.2.4 Transactions and blockchains

The semantics for blockchains is given as a transition system defined by the rules given in Figure 6. Here, the rule [GENESIS] describes the “genesis event” where contracts are declared, whilst [TRANS] describes a single transaction. This is thus a *small-step* semantics, invoking the big-step semantics for declarations and statements for its premises. We remark that the rules of the operational semantics for blockchains (as well as those for statements presented above) define a deterministic transition relation.

Note that, unlike in the original formulation of TINY SOL, we do not include a rule like [Tx2] in [3] for rolling back a transaction in case it is non-terminating or it aborts via a `throw` command. Such a rule would require a premise that cannot be checked effectively for a Turing-complete language like TINY SOL and therefore we omit it, since it is immaterial for the main contributions we give in this paper.² In practice, termination of Ethereum smart contracts is ensured via a “gas mechanism” and is assumed by techniques for the formal analysis of smart contracts. However, as observed in, for instance, [11], proof of termination for smart contracts is non-trivial even in the presence of a “gas mechanism.” In the aforementioned paper, the authors present the first mechanised proof of termination of contracts written in EVM bytecode using minimal assumptions on the gas cost of operations (see the study [29] for an empirical analysis of the effectiveness of the “gas mechanism” in estimating the computational cost of executing real-life transactions). We leave for future work the addition of a “gas mechanism” to TINY SOL and the adaption of the results we present in this paper to that setting.

² For instance, rule [Tx2] in [3] has an undecidable premise that checks whether the execution of the body of a contract does *not* yield a final state. It is debatable whether such rules should appear in an operational semantics.

3 Call integrity and noninterference in TinySol

Grishchenko et al. [13] formulate the property of *call integrity* for smart contracts written in the language EVM, which is the “low-level” bytecode of the Ethereum platform, and the target language to which e.g. Solidity compiles. They then prove [13, Theorem 1] that this property suffices for ruling out reentrancy phenomena, as those described in the example in Figure 1. We first formulate a similar property for TINY SOL; this requires a few preliminary definitions.

► **Definition 3** (Trace semantics). A trace of method invocations is given by

$$\pi ::= \epsilon \mid X \rightarrow Y . f(\tilde{v}) : n, \pi$$

where X is the address of the calling contract, Y is the address of the called contract, f is the method name, and \tilde{v} and n are the actual parameters.

We annotate the big-step semantics with a trace containing information on the invoked methods to yield labeled transitions of the form $\xrightarrow{\pi}_S$. To do this, we modify the rules in Table 5 as follows:

- in rules [BS-SKIP], [BS-LOOP_F], [BS-AssV] and [BS-AssF], every occurrence of \rightarrow_S becomes $\xrightarrow{\epsilon}_S$;
- in rules [BS-IF] and [BS-DECV], every occurrence of \rightarrow_S becomes $\xrightarrow{\pi}_S$;
- rules [BS-SEQ], [BS-LOOPT] and [BS-CALL] respectively become:

$$\frac{\begin{array}{c} \text{env}_T \vdash \langle S_1, \text{env}_{SV} \rangle \xrightarrow{\pi_1}_S \text{env}_{SV}'' \\ \text{env}_T \vdash \langle S_2, \text{env}_{SV}'' \rangle \xrightarrow{\pi_2}_S \text{env}'_{SV} \end{array}}{\text{env}_T \vdash \langle S_1 ; S_2, \text{env}_{SV} \rangle \xrightarrow{\pi_1, \pi_2}_S \text{env}'_{SV}} \quad \frac{\begin{array}{c} \text{env}_{SV} \vdash e \rightarrow_e T \\ \text{env}_T \vdash \langle S, \text{env}_{SV} \rangle \xrightarrow{\pi_1}_S \text{env}_{SV}'' \\ \text{env}_T \vdash \langle \text{while } e \text{ do } S, \text{env}_{SV}'' \rangle \xrightarrow{\pi_2}_S \text{env}'_{SV} \end{array}}{\text{env}_T \vdash \langle \text{while } e \text{ do } S, \text{env}_{SV} \rangle \xrightarrow{\pi_1, \pi_2}_S \text{env}'_{SV}}$$

$$\frac{\dots \quad \text{env}_T \vdash \langle S, \text{env}_{SV}'' \rangle \xrightarrow{\pi}_S \text{env}'_{SV}}{\text{env}_T \vdash \langle e_1 . f(\tilde{e}) : e_2, \text{env}_{SV} \rangle \xrightarrow{X \rightarrow Y . f(\tilde{v}) : n, \pi} \text{env}'_S, \text{env}_V}$$

The full definition is given in [1]. We extend this annotation to the semantics for blockchains and write $\xrightarrow{\pi}_B$ for this annotated relation.

► **Definition 4** (Projection). The projection of a trace to a specific contract X , written $\pi \downarrow_X$, is the trace of calls with X as the calling address. Formally:

$$\epsilon \downarrow_X = \epsilon \quad (Z \rightarrow Y . f(\tilde{v}) : n, \pi) \downarrow_X = \begin{cases} X \rightarrow Y . f(\tilde{v}) : n, (\pi \downarrow_X) & \text{if } Z = X \\ \pi \downarrow_X & \text{otherwise} \end{cases}$$

Notationally, given a (partial) function f , we write $f|_X$ for denoting the restriction of f to the subset X of its domain.

► **Definition 5** (Call integrity). Let \mathcal{A} denote the set of all contracts (addresses), $\mathcal{X} \subseteq \mathcal{A}$ denote a set of trusted contracts, $\mathcal{Y} \triangleq \mathcal{A} \setminus \mathcal{X}$ denote all other contracts, and $\text{env}_{ST}^{\mathcal{X}}$ have domain \mathcal{X} . A contract $C \in \mathcal{X}$ has call integrity for \mathcal{Y} if, for every transaction T and environments env_{ST}^1 and env_{ST}^2 such that $\text{env}_{ST}^1(X)|_{\mathcal{X}} = \text{env}_{ST}^2(X)|_{\mathcal{X}} = \text{env}_{ST}^{\mathcal{X}}$, it holds that

$$\langle T, \text{env}_{ST}^1 \rangle \xrightarrow{\pi_1}_B \text{env}_{ST}^{1'} \wedge \langle T, \text{env}_{ST}^2 \rangle \xrightarrow{\pi_2}_B \text{env}_{ST}^{2'} \implies \pi_1 \downarrow_C = \pi_2 \downarrow_C$$

The definition is quite complicated and contains a number of elements:

- C is the contract of interest.

- \mathcal{X} is a set of *trusted* contracts, which we assume are allowed to influence the behaviour of C . This set must obviously contain C , since C at least must be assumed to be trusted. Thus, a contract C can have call integrity for all contracts, if $\mathcal{X} = \{C\}$.
- Conversely, the set $\mathcal{Y} = \mathcal{A} \setminus \mathcal{X}$ is the set of addresses of all contracts that are *untrusted*.³
- env_{ST}^1 and env_{ST}^2 are any two pairs of method/field environments that coincide (both in the code and in the values) for all the trusted contracts.⁴ The point is that the contracts in \mathcal{X} are assumed to be known, and hence invariant, whereas any contract in \mathcal{Y} is assumed to be unknown and may be controlled by an attacker. Thus, we are actually quantifying over all possible contexts where the contracts in \mathcal{X} can be run.
- T is any transaction; it may be issued from any account and to any contract. Thus we also quantify over all possible transactions, since an attacker may request an arbitrary transaction, that is thus part of the execution context as well.

Then, the call integrity property intuitively requires that, if we run the trusted part of the code in any execution context, the behavior of C remains the same, i.e. C must make exactly the same method calls (and in exactly the same order). Thus, to *disprove* that C has call integrity, it suffices to find two environments and a transaction that will induce a difference in the call trace of C .

The idea in the property of call integrity is that the behaviour of C should not depend on any untrusted code (i.e. contracts in \mathcal{Y}), even if control is transferred to a contract in \mathcal{Y} . The latter could for example happen if C calls a method on $B \in \mathcal{X}$, and B then calls a method on a contract in \mathcal{Y} . This also means that C cannot directly call any contract in \mathcal{Y} , since that can only happen if C calls a method on a contract, where the address is received as a parameter, or if it calls a method on a “hard-coded” contract address. In both cases, we can easily pick up two environments able to induce different behaviors, for example by choosing a non-existing address for one context (in the first case), or by ensuring that no contract exists on the hard-coded address in one context (in the second case). The latter possibility can seem somewhat contrived, especially if we assume that all contracts are created at the genesis event, and it might therefore be reasonable to require also that $\text{dom}(\text{env}_{ST}^1) = \text{dom}(\text{env}_{ST}^2)$, such that we at least assume that contracts exist on the same addresses. However, on an actual blockchain, new contracts can be deployed (and in some cases also deleted) at any time, and if such a degree of realism is desired, this extra constraint should not be imposed.

The main problem with the definition of call integrity is that it relies on a universal quantification over all possible executions contexts. This makes it hard to be checked in practice. However, our previous discussion indicates that call integrity may intuitively be viewed as a form of *noninterference* between the trusted and the untrusted contracts. We now see to what extent this intuition is true and formally compare the two notions.

First of all, we consider a basic lattice of security levels, made up by just two levels, namely H (for *high*) and L (for *low*), with $L < H$. We tag every contract to be high or low through a contracts-to-levels mapping $\lambda : \mathcal{A} \rightarrow \{L, H\}$; this induces a bipartition of the contract names \mathcal{A} into the following sets:

$$\mathcal{L} = \{X \in \mathcal{A} \mid \lambda(X) = L\} \quad \mathcal{H} = \{X \in \mathcal{A} \mid \lambda(X) = H\}$$

³ Note that this is formulated inversely by Grishchenko et al., who instead formulate the property for a set of *untrusted* contracts \mathcal{A}_C , corresponding to \mathcal{Y} in the present formulation. However, using the set of *trusted* addresses \mathcal{X} seems more straightforward.

⁴ This too is inversely formulated by Grishchenko et al.

In this way, we create a bipartition of the state into low and high, corresponding to the fields of the low and of the high contracts, respectively. Then, we define *low-equivalence* $=_L$ to be the equivalence on states such that $\text{env}_S^1 =_L \text{env}_S^2$ if and only if $\text{env}_S^1(X) = \text{env}_S^2(X)$, for every $X \in \mathcal{L}$.

We can now adapt the notion of noninterference for multi-threaded programs by Smith and Volpano [25] to the setting of TINY SOL.

► **Definition 6** (Noninterference). *Given a contracts-to-levels mapping $\lambda : \mathcal{A} \rightarrow \{L, H\}$ and a contract environment env_T , the contracts satisfy noninterference if, for every env_S^1 and env_S^2 and for every transaction T such that*

$$\text{env}_S^1 =_L \text{env}_S^2 \quad \langle T, \text{env}_S^1, \text{env}_T \rangle \rightarrow_B \text{env}_S^{1'}, \text{env}_T \quad \langle T, \text{env}_S^2, \text{env}_T \rangle \rightarrow_B \text{env}_S^{2'}, \text{env}_T$$

it holds that $\text{env}_S^{1'} =_L \text{env}_S^{2'}$.

► **Remark 7** (Incomparability). Call integrity and noninterference seem strongly related, in the sense that the first requires that the behaviour of a contract is not influenced by the (bad) execution context, whereas the second one requires that a part of the computation (the “low” one) is not influenced by the remainder context (the “high” one). So, one may try to prove a statement like: “ $C \in \mathcal{X}$ has call integrity for $\mathcal{Y} \triangleq \mathcal{A} \setminus \mathcal{X}$ if and only if it satisfies noninterference w.r.t. λ such that $\mathcal{L} = \mathcal{X}$ and $\mathcal{H} = \mathcal{Y}$.” However, both directions are false.

For the direction from right to left, consider:

```

1 contract X {
2   field balance = 0
3   go() { }
4 }
```

```

contract Y {
  field balance = v
  go() { X.go():this.balance }
```

where **X** is trusted and **Y** untrusted. Since **X** cannot invoke any method, this example satisfies call integrity. However, it does not satisfy noninterference. To see this, consider two environments, one assigning 1 to **Y**’s balance and the other one assigning 0, and the transaction **Y->Y.go():0**.

For the direction from left to right, consider the following:

```

1 contract X {
2   go() {
3     if Y.balance = 0
4       then Z.a():0
5       else Z.b():0
6   }
7 }
```

```

contract Y {
  field balance = v;
}
contract Z {
  a() { }
  b() { }
}
```

Assuming that both **X** and **Z** are low, the example satisfies noninterference: there is no way for **Y** to influence the low memory. By contrast, the code does not satisfy call integrity. Indeed, let **v** be 0 in one environment and 1 in the other, and consider **T** to be **X->X.go():0**: in the first environment, it generates **X->Z.a():0**, whereas in the second one it generates **X->Z.b():0**.

4 A type system for noninterference and call integrity

As demonstrated in Remark 7, call integrity and noninterference are incomparable properties. This is so because noninterference is a 2-property on the pair of *stores* $(\text{env}_S^{1'}, \text{env}_S^{2'})$ resulting from two different executions, whereas call integrity is a 2-property on the pair of *call traces*

(π_1, π_2) generated during two executions. However, the two properties have an interesting overlap, because an outgoing currency flow (i.e. a method call) may also result, at least potentially, in a change of the stored values of the `balance` fields of the sender and recipient. Every method call is therefore *also* an information flow between the two, even when no amount of currency is transferred. In [28], Volpano et al. devise a type system for checking information flows, which, as they show, yields a sound approximation to noninterference. In the following, we create an adaptation of this type system to TINY SOL and show that it may *also* be used to soundly approximate call integrity.

4.1 Type syntax

We begin by assuming a finite lattice $(\mathcal{S}, \sqsubseteq)$ consisting of a set of *security levels* \mathcal{S} , ranged over by s , and equipped with a partial order \sqsubseteq . We write s_{\perp} , and s_{\top} for the least and largest elements in \mathcal{S} .

In the simplest setting, we can let $\mathcal{S} \stackrel{\Delta}{=} \{L, H\}$ (for “low” and “high”) and define $L \sqsubseteq L$, $L \sqsubseteq H$, and $H \sqsubseteq H$. This is sufficient for ensuring bi-partite noninterference, but the type system can also handle more fine-grained security control. With this, we can define the types:

► **Definition 8.** *We use the following language of types, where $I \in \text{TNames}$ is a type name (or “interface name”):*

$$\begin{array}{ll} B \in \mathcal{B} ::= s \mid I_s & T \in \mathcal{T} ::= B \mid \text{var}(B) \mid \text{cmd}(s) \mid \text{proc}(B) : s \\ \Gamma \in \mathcal{G} ::= \mathcal{N} \multimap \mathcal{T} \cup \mathcal{G} & \mathcal{N} ::= \text{ANames} \cup \text{FNames} \cup \text{VNames} \cup \text{MNames} \cup \text{TNames} \end{array}$$

We write \tilde{T} for a tuple of types (T_1, \dots, T_n) .

Note that for the purpose of the type system, unless otherwise noted, we shall assume that the four “magic names” `MVar` are contained in the respective sets of field and variable names; i.e. `balance` \in `FNames` and `this`, `sender`, `value` \in `VNames`.

The meaning of the types is as follows:

- \mathcal{B} is a set of *base types*, which can either be a security level s , or an interface name I , annotated with a security level, I_s . Security levels are assigned to plain data, i.e. *values* of type `int` or `bool`, as well as *expressions* yielding values of these types. The annotated interface type is assigned to *addresses*, as well as expressions yielding addresses. In either case, the meaning of the type s (resp. I_s), when given to an expression e , is that all variables *read from* within e , are of level s or lower.

Note that for the purpose of the present type system, we do not distinguish between values of type `int` and `bool`, in the sense that we do not check whether these type constraints are preserved. Instead, we shall just assume that all programs are well-typed w.r.t. these simple type constraints, such that e.g. expressions in the guards of `if` and `while` constructs indeed yield boolean values. The present type system can easily be extended to incorporate such a simple type check by extending the set of base types with annotated value types `ints` and `bools`, similar to the annotated interface types.

- $\text{var}(B)$ is a box type given to value *containers*, i.e. variables and fields. It denotes that the container can store data of type B . In the case of $\text{var}(s)$, it denotes that the box can store data of level s or lower, whereas in the case of $\text{var}(I_s)$ it additionally denotes that the address stored in the variable must be of type I .
- $\text{cmd}(s)$ is a *phrase type* given to *code*, i.e. commands S . It denotes that all *assignments* in the code are made to variables whose security level is s or higher.

- $\text{proc}(\tilde{B}):s$ is a procedure type given to methods $f(\tilde{x}) \{ S \}$. It denotes that the body S can be typed as $\text{cmd}(s)$, under the assumption that the formal parameters \tilde{x} have types $\text{var}(\tilde{B})$. We shall discuss the types assigned to the “magic variables” `this`, `sender` and `value` below.

Note that every method declaration contains an implicit write to the `balance` field of the containing contract: hence, given the meaning of $\text{cmd}(s)$, this also means that the security level of `balance` must always be s or higher than the level of any method declared in an interface.

Finally, Γ is a type environment, which is a partial function from names to types *or type environments*. The latter possibility is included because we shall represent each contract declaration as its own type environment, containing box types and procedure types for the fields and methods of the contract, and pointed to by the corresponding interface name. Thus, if a contract has address X , then $\Gamma(X) = I_s$ for some interface name I and security level s , and $\Gamma(I) = \Gamma_I$, where Γ_I is a type environment containing the signatures of the methods and fields of the contract. We shall use the following simple interface declaration language for the interfaces of contracts:

```
 $IC ::= \epsilon \mid \text{interface } I \{ IF \; IM \} \; IC$ 
 $IF ::= \epsilon \mid \text{field } p : \text{var}(B); \; IF$ 
 $IM ::= \epsilon \mid \text{method } f : \text{proc}(B):s; \; IM$ 
```

mirroring the syntax of contract declarations.

We require that all interface declarations be *well-formed* in the sense that they must at least contain a declaration for the mandatory members, i.e. the `balance` field and the `send()` method. This ensures that we can define a minimal interface declaration called I^\top , such that every well-formed interface declaration is a specialisation of I^\top . This minimal interface contains just the signatures of the mandatory `balance` field and of the `send()` method; i.e.

```
 $1 \; \text{interface } I^\top \{$ 
 $2 \; \; \text{field } \text{balance} : \text{var}(s_\top);$ 
 $3 \; \; \text{method } \text{send} : \text{proc}():s_\perp;$ 
 $4 \; \}$ 
```

in the aforementioned interface declaration syntax.

Intuitively, this definition ensures that, for any valid interface definition I (containing at least `balance` and `send`) and any security level annotation s , it must hold that I_s is a subtype of $I_{s_\top}^\top$, thus always allowing us to type I_s up to $I_{s_\top}^\top$. In the following section, we shall give a definition of a subtyping relation that will ensure that this indeed is the case.

The inclusion of a contract “supertype” $I_{s_\top}^\top$ is similar to what is done in the type system developed for Featherweight Solidity by Crafa et al. in [7]. This is necessary to enable us to give a type to the “magic variable” `sender`, which is available within the body of every method, since this variable can be bound to the address of any contract or account. We shall assume that $I^\top \in \text{dom}(\Gamma)$ for any Γ we shall consider.

We shall also use a *typed* syntax of TINY SOL, where local variables are now declared as

$\text{var}(B) \; x := e$

where B is the type of the value of the expression e . Likewise, we add annotated type names I_s to contract declarations thus:

$\text{contract } X : I_s \{ DF; DM \}$

where I is a declared type name. Note that the security level is given on the *contract*, rather than on the interface. This is intentional, since multiple contracts may implement the

$$\begin{array}{c}
[\text{SUBS-NAME}] \frac{\Gamma \vdash \Gamma(I^1) <: \Gamma(I^2)}{\Gamma \vdash I_{s^1}^1 <: I_{s^2}^2} (s_1 \sqsubseteq s_2) \quad [\text{SUBS-ENV}] \frac{\forall n \in \text{dom}(\Gamma_2). \Gamma_1(n) <: \Gamma_2(n)}{\Gamma \vdash \Gamma_1 <: \Gamma_2} (\text{dom}(\Gamma_2) \subseteq \text{dom}(\Gamma_1)) \\
[\text{SUBS-SEC}] \frac{}{\Gamma \vdash s_1 <: s_2} (s_1 \sqsubseteq s_2) \quad [\text{SUBS-CMD}] \frac{}{\Gamma \vdash cmd(s_1) <: cmd(s_2)} (s_2 \sqsubseteq s_1) \\
[\text{SUBS-VAR}] \frac{\Gamma \vdash B_1 <: B_2}{\Gamma \vdash var(B_1) <: var(B_2)} \quad [\text{SUBS-PROC}] \frac{\Gamma \vdash \tilde{B}_1 <: \tilde{B}_2}{\Gamma \vdash proc(\tilde{B}_1):s_1 <: proc(\tilde{B}_2):s_2} (s_2 \sqsubseteq s_1)
\end{array}$$

Figure 7 Subtyping rules.

same interface but nevertheless be categorised into different security levels. For the sake of simplicity, we shall omit the explicit definition of interfaces in the code and merely assume that an interface declaration Γ_I with an associated name I is provided for each contract.

4.2 Subtyping

We shall introduce a parametrised subtyping relation $\Gamma \vdash \cdot <: \cdot$ on types. For each choice of Γ , we define it as the least preorder satisfying the rules given in Figure 7. The parameter Γ is needed to handle subtyping for interface names I in rule [SUBS-NAME]. Note that by this rule we have, for each well-formed interface I_s (with security level s and interface name I) declared in Γ , that $\Gamma \vdash I_s <: I_{s^\top}^\top$ as expected. Also note that we write $\Gamma \vdash \tilde{B}_1 <: \tilde{B}_2$ to mean $\Gamma \vdash B_1^i <: B_2^i$ for each i ($1 \leq i \leq n$, where $|\tilde{B}_1| = n = |\tilde{B}_2|$).

By rule [SUBS-SEC], subtyping is covariant in the types of *data*, i.e. the security level s , and likewise, the box type constructor $var(B)$ is covariant by rule [SUBS-VAR]. On the other hand, the type constructor for commands, $cmd(s)$, is *contravariant* by rule [SUBS-CMD]. Lastly, the type constructor for methods, $proc(\tilde{B}):s$, is covariant in the input parameters \tilde{B} by rule [SUBS-PROC], but contravariant in the “return” type s , which indicates the level of the underlying command type. These variances are consistent with the intended meaning of the types:

- A box of type $var(B)$ can store something of B or lower (where B is either s or I_s). Hence, if $\Gamma \vdash B_1 <: B_2$, then a box type $var(B_2)$ can safely be used wherever a box type $var(B_1)$ is needed.
- A command of type $cmd(s)$ will assign to variables whose level is s or higher. Hence, if $s_1 \sqsubseteq s_2$, then a command type $cmd(s_1)$ can safely be used wherever a command type $cmd(s_2)$ is needed.
- A method of type $proc(\tilde{B}):s$ expects parameters of types \tilde{B} and promises that the method body will only assign to variables that are level s or higher. Hence, if $\Gamma \vdash \tilde{B}_1 <: \tilde{B}_2$ and $s_2 \sqsubseteq s_1$, a command type $proc(\tilde{B}_2):s_2$ can safely be used wherever a command type $proc(\tilde{B}_1):s_1$ is needed. This is consistent with the type for the body S since, if S can be typed to level $cmd(s_1)$, then it can also safely be typed to level $cmd(s_2)$.

4.3 Type judgments

We can now give the rules for concluding type judgments, starting with the type rules for declarations given in Figure 8.

Type judgments for contract declarations are of the form $\Gamma \vdash DC$, stating that the declarations DC are *well-typed* w.r.t. the environment Γ . This holds if the declarations are consistent with the type information recorded in Γ , i.e. every field and method must have a

$$\begin{array}{c}
 [T\text{-DEC-C}] \frac{\Gamma(X) = I_s \quad \Gamma_1 = \Gamma, \mathbf{this} : var(I_s) \quad \Gamma \vdash DC \quad \Gamma_1 \vdash DF \quad \Gamma_1 \vdash DM}{\Gamma \vdash \mathbf{contract} \ X : I_s \not\models DF \ DM \models DC} \\
 [T\text{-DEC-F}] \frac{\Gamma(\mathbf{this}) = var(I_s) \quad p \in \text{dom } (\Gamma(I)) \quad \Gamma \vdash DF}{\Gamma \vdash \mathbf{field} \ p := v; \ DF} \\
 [T\text{-DEC-M}] \frac{\begin{array}{c} \Gamma(\mathbf{this}) = var(I_{s_1}) \quad \Gamma(I)(f) = proc(\tilde{B}):s \\ \Gamma_1 = \Gamma, \tilde{x} : var(\tilde{B}), \mathbf{value} : var(s), \mathbf{sender} : var(I_{s_1}^\top) \\ \Gamma \vdash \mathbf{this.balance} : var(s) \quad \Gamma_1 \vdash S : cmd(s) \quad \Gamma \vdash DM \end{array}}{\Gamma \vdash f(\tilde{x}) \not\models S \models DM}
 \end{array}$$

■ **Figure 8** Type rules for declarations.

$$\begin{array}{c}
 [T\text{-ENV-T}] \frac{\Gamma, \mathbf{this} : \Gamma(X) \vdash \mathbf{env}_M \quad \Gamma \vdash \mathbf{env}_T}{\Gamma \vdash \mathbf{env}_T, (X, \mathbf{env}_M)} \\
 [T\text{-ENV-M}] \frac{\begin{array}{c} \Gamma(\mathbf{this}) = var(I_{s_1}) \quad \Gamma(I)(f) = proc(\tilde{B}):s \\ \Gamma_1 = \Gamma, \tilde{x} : var(\tilde{B}), \mathbf{value} : var(s), \mathbf{sender} : var(I_{s_1}^\top) \\ \Gamma \vdash \mathbf{env}_M \quad \Gamma \vdash \mathbf{this.balance} : var(s) \quad \Gamma_1 \vdash S : cmd(s) \end{array}}{\Gamma \vdash \mathbf{env}_M, (f, (\tilde{x}, S))} \\
 [T\text{-ENV-S}] \frac{\Gamma, \mathbf{this} : \Gamma(X) \vdash \mathbf{env}_F \quad \Gamma \vdash \mathbf{env}_S}{\Gamma \vdash \mathbf{env}_S, (X, \mathbf{env}_F)} \\
 [T\text{-ENV-F}] \frac{\Gamma(\mathbf{this}) = var(I_s) \quad p \in \text{dom } (\Gamma(I)) \quad \Gamma \vdash \mathbf{env}_F}{\Gamma \vdash \mathbf{env}_F, (p, v)} \\
 [T\text{-ENV-V}] \frac{\Gamma \vdash \mathbf{env}_V}{\Gamma \vdash \mathbf{env}_V, (x, v)} \ (x \in \text{dom } (\Gamma))
 \end{array}$$

■ **Figure 9** Type rules for environment agreement.

type, and the body of each method must be typable according to the assumptions of the type. Note that the check here only ensures that every declared contract member has a type; the converse check (i.e. that every declared type in an interface also has an implementation) should also be performed. However, we shall omit this in the present treatment.

After the initial reduction step, all declarations are stored in the two environments \mathbf{env}_{ST} , and further reductions also use the variable environment \mathbf{env}_V for local variable declarations. Hence, we also need to be able to conclude *agreement* between these environments and Γ . These rules are given in Figure 9, closely mirroring those of Figure 8. We omit the type rules for empty environments (since an empty environment is always well-typed). As with declarations above, we also omit the rules for ensuring that all declared types in an interface also have an implementation in any contract claiming to implement that interface.

Next, we consider the type rules for statements appearing in the body of method declarations; they are given in Figure 10. Here, judgments are of the form $\Gamma \vdash S : cmd(s)$, indicating that s is the *lowest* level of any variable written to within S . This is derived from the types of the variables occurring in S , i.e. the types $var(B)$. However, as B can be either s or I_s , we need a way to extract just the security level and drop the interface name. For this, we write $B \rightsquigarrow s$, defined in the obvious way:

$$s \rightsquigarrow s \qquad I_s \rightsquigarrow s$$

1:16 A Sound Type System for Secure Currency Flow

$$\begin{array}{c}
[\text{T-SKIP}] \frac{}{\Gamma \vdash \text{skip} : cmd(s_{\top})} \quad [\text{T-SUBS-S}] \frac{\Gamma \vdash S : cmd(s_1) \quad \Gamma \vdash cmd(s_1) <: cmd(s_2)}{\Gamma \vdash S : cmd(s_2)} \\
\\
[\text{T-THROW}] \frac{}{\Gamma \vdash \text{throw} : cmd(s_{\top})} \quad [\text{T-DECVAR}] \frac{\Gamma \vdash e : B \quad \Gamma, x : var(B) \vdash S : cmd(s)}{\Gamma \vdash var(B) \ x := e \text{ in } S : cmd(s)} \\
\\
[\text{T-ASS-V}] \frac{\Gamma \vdash x : var(B) \quad \Gamma \vdash e : B}{\Gamma \vdash x := e : cmd(s)} \quad (B \rightsquigarrow s) \quad [\text{T-ASS-F}] \frac{\Gamma \vdash e_1.p : var(B) \quad \Gamma \vdash e_2 : B}{\Gamma \vdash e_1.p := e_2 : cmd(s)} \quad (B \rightsquigarrow s) \\
\\
[\text{T-SEQ}] \frac{\Gamma \vdash S_1 : cmd(s) \quad \Gamma \vdash S_2 : cmd(s)}{\Gamma \vdash S_1 ; S_2 : cmd(s)} \quad [\text{T-CALL}] \frac{\Gamma \vdash e_1.f : proc(\tilde{B}) : s \quad \Gamma \vdash \text{this.balance} : var(s) \quad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1.f(\tilde{e}) : e_2 : cmd(s)} \\
\\
[\text{T-LOOP}] \frac{\Gamma \vdash e : s \quad \Gamma \vdash S : cmd(s)}{\Gamma \vdash \text{while } e \text{ do } S : cmd(s)} \quad [\text{T-IF}] \frac{\Gamma \vdash e : s \quad \Gamma \vdash S_{\top} : cmd(s) \quad \Gamma \vdash S_F : cmd(s)}{\Gamma \vdash \text{if } e \text{ then } S_{\top} \text{ else } S_F : cmd(s)}
\end{array}$$

Figure 10 Type rules for statements.

$$\begin{array}{c}
[\text{T-VAR}] \frac{\Gamma \vdash x : var(B)}{\Gamma \vdash x : B} \quad [\text{T-VAL}] \frac{}{\Gamma \vdash v : B} \left(B = \begin{cases} \Gamma(v) & \text{if } v \in \text{ANames} \\ s & \text{otherwise} \end{cases} \right) \\
\\
[\text{T-FIELD}] \frac{\Gamma \vdash e.p : var(B)}{\Gamma \vdash e.p : B} \quad [\text{T-SUBS-E}] \frac{\Gamma \vdash e : B_1 \quad \Gamma \vdash B_1 <: B_2}{\Gamma \vdash e : B_2} \\
\\
[\text{T-OP}] \frac{\Gamma \vdash e_1 : B_1 \quad \dots \quad \Gamma \vdash e_n : B_n}{\Gamma \vdash \text{op}(e_1, \dots, e_n) : s} \left(\begin{array}{c} B_1 \rightsquigarrow s \\ \vdots \\ B_n \rightsquigarrow s \end{array} \right)
\end{array}$$

Figure 11 Type rules for expressions.

This is used in the rules for assignments (rules [T-ASS-V] and [T-ASS-F]). Note that in the rules [T-IF] and [T-LOOP], we know (by our assumption that all contracts are well-typed w.r.t. simple type preservation) that e will evaluate to a boolean value, which therefore necessarily will have a type s . Thus, we do not need the extra step of $B \rightsquigarrow s$ here.

All rules are straightforward, except for [T-CALL]. According to the semantics for call (cf. rule [BS-CALL]), every call includes an implicit read and write of the `balance` field of the calling contract, since the call will only be performed if the value of e_2 is less than, or equal to, the value of `balance` (to ensure that the subtraction will not yield a negative number). There is thus an implicit flow from `this.balance` to the body S of the method call, similar to the case for the guard expression e in an if-statement. Furthermore, there is an implicit write to the `balance` field of the callee, and thus a flow of information from one field to the other. This might initially seem like it would require both caller and callee to have the same security level for their `balance` field. However, the levels *can* differ, since by subtyping we can coerce one up to match the level of the other. For this reason, we have $\Gamma \vdash \text{this.balance}$ in the premise, to be explicitly *concluded*, rather than as a simple lookup. This enables calls from a lower security level into a higher security level, but not the other way around.

Next, we consider the type rules for expressions e , given in Figure 11. Here, judgments are of the form $\Gamma \vdash e : B$. There are a few things to note:

- In rule [T-VAL], the type of a value v can be chosen freely, if v is a value type, i.e. of type `int` or `bool`. This rule is a consequence of the fact that there is no simple relationship between the datatype of a value and its security level. The actual security level will then be determined by the type of the variable (resp. field) to which it is assigned.

$$\begin{array}{c}
 [\text{T-BOX-X}] \frac{}{\Gamma \vdash x : \text{var}(B)} (\Gamma(x) = \text{var}(B)) \quad [\text{T-BOX-F}] \frac{\Gamma \vdash e : I_s}{\Gamma \vdash e.p : \text{var}(B)} \left(\begin{array}{l} \Gamma(I)(p) = \text{var}(B) \\ B \rightsquigarrow s \end{array} \right) \\
 [\text{T-M-SUB}] \frac{\begin{array}{c} \Gamma \vdash e.f : \text{proc}(\tilde{B}_1) : s_1 \\ \Gamma \vdash \text{proc}(\tilde{B}_1) : s_1 <: \text{proc}(\tilde{B}_2) : s_2 \end{array}}{\Gamma \vdash e.f : \text{proc}(\tilde{B}_2) : s_2} \quad [\text{T-METH}] \frac{\Gamma \vdash e : I_s}{\Gamma \vdash e.f : \text{proc}(\tilde{B}) : s} \left(\Gamma(I)(f) = \text{proc}(\tilde{B}) : s \right)
 \end{array}$$

■ **Figure 12** Type rules for method, variable and field lookup.

- The rules [T-VAR] and [T-FIELD] simply unwrap the type of the contained value from the box type of the container. Note that here we assume that x also covers the “magic variable” names `this`, `sender` and `value`, and that p also covers the field name `balance`.
- Finally, in rule [T-OP], we require that all arguments and the return value must be typable to the same security level s . Note in particular that we assume that *no* operation is defined with an address *return* type; i.e. we do not allow any form of pointer arithmetic. Operations may be defined on addresses for their *arguments*, e.g. equality testing, but the return type must be one of the other value types, which can be given a security level. Thus, in the rule [T-OP], we also need to extract the security level s from the types of the argument expressions.

Finally, we have the look-up rules for methods, variables and fields, given in Figure 12.

- In rule [T-BOX-X] we assume that x also covers the magic variable names `this`, `sender` and `value`.
- In rule [T-BOX-F] we assume that p also covers the special field name `balance`. Furthermore, we require e in $e.p$ to resolve to an *interface name* rather than variable; i.e. the expression must yield an address. This is again warranted by our assumption that expressions are well-typed w.r.t. simple type preservation.
- The same is the case in rule [T-METH] for method lookup $e.f$, which is used in the premise of the rule [T-CALL].

In the lookup rules, the expression e is an *object path*, which must resolve to an address. As we disallow operations op to return addresses, the object paths form a proper subset of the set of expressions, since they can only consist of variable lookups, field reads or addresses given as pure values. Note that, in the rules [T-BOX-F] and [T-METH], we require that the object path e must be typable as an interface with the *same* security level s as the value (resp. method) that is being looked up. This is necessary to ensure that values residing in a higher-level part of the memory cannot affect values at lower levels, in this case by altering the *path* to the object being resolved.

4.4 Safety and soundness

As is the case for the type system proposed in [28], our type system does not have a now-safety predicate in the usual sense, since (invariant) safety in simple type systems is a 1-property, whereas noninterference is a hyper-property (specifically, a 2-property). Instead, the meaning of “safety” is expressed directly in the meaning of the types. Specifically:

- If an expression e has type B such that $B \rightsquigarrow s$, then it denotes that all variables *read from* in the evaluation of e are of level s or *lower*, i.e. no read-up.
- If a statement S has type $\text{cmd}(s)$, then it denotes that all variables *written to* in the execution of S are of level s or *higher*, i.e. no write-down.

$$\begin{aligned}
 & [\text{EQ-ENV-EMPTY}] \frac{}{\Gamma \vdash \mathsf{env}_X^\emptyset =_s \mathsf{env}_X^\emptyset} (X \in \{V, S, F, T, M\}) \\
 & [\text{EQ-ENV}_V] \frac{\Gamma \vdash \mathsf{env}_V^1 =_s \mathsf{env}_V^2}{\Gamma \vdash \mathsf{env}_V^1, (x, v_1) =_s \mathsf{env}_V^2, (x, v_2)} \left(\begin{array}{l} \Gamma(x) = \mathit{var}(s') \\ s' \sqsubseteq s \implies v_1 = v_2 \end{array} \right) \\
 & [\text{EQ-ENV}_S] \frac{\Gamma \vdash \mathsf{env}_S^1 =_s \mathsf{env}_S^2 \quad \Gamma(\Gamma(X)) \vdash \mathsf{env}_F^1 =_s \mathsf{env}_F^2}{\Gamma \vdash \mathsf{env}_S^1, (X, \mathsf{env}_F^1) =_s \mathsf{env}_S^2, (X, \mathsf{env}_F^2)} \\
 & [\text{EQ-ENV}_F] \frac{\Gamma \vdash \mathsf{env}_F^1 =_s \mathsf{env}_F^2}{\Gamma \vdash \mathsf{env}_F^1, (p, v_1) =_s \mathsf{env}_F^2, (p, v_2)} \left(\begin{array}{l} \Gamma(p) = \mathit{var}(s') \\ s' \sqsubseteq s \implies v_1 = v_2 \end{array} \right) \\
 & [\text{EQ-ENV}_T] \frac{\Gamma \vdash \mathsf{env}_T^1 =_s \mathsf{env}_T^2}{\Gamma \vdash \mathsf{env}_T^1, (X, \mathsf{env}_M^1) =_s \mathsf{env}_T^2, (X, \mathsf{env}_M^2)} \left(\begin{array}{l} \Gamma(X) = I_{s'} \\ s' \sqsubseteq s \implies \mathsf{env}_M^1 = \mathsf{env}_M^2 \end{array} \right) \\
 & [\text{EQ-ENV}_{SV}] \frac{\Gamma \vdash \mathsf{env}_S^1 =_s \mathsf{env}_S^2 \quad \Gamma \vdash \mathsf{env}_V^1 =_s \mathsf{env}_V^2}{\Gamma \vdash \mathsf{env}_{SV}^1 =_s \mathsf{env}_{SV}^2} \\
 & [\text{EQ-ENV}_{ST}] \frac{\Gamma \vdash \mathsf{env}_S^1 =_s \mathsf{env}_S^2 \quad \Gamma \vdash \mathsf{env}_T^1 =_s \mathsf{env}_T^2}{\Gamma \vdash \mathsf{env}_{ST}^1 =_s \mathsf{env}_{ST}^2}
 \end{aligned}$$

 **Figure 13** Rules for the s -parameterised equivalence relation.

Intuitively, the meaning of these two types together imply that information from higher-level variables cannot flow into lower-level variables. For a statement such as $x := e$ to be well-typed, it must therefore be the case that, if $\Gamma \vdash x : \mathit{var}(s_1)$ and $\Gamma \vdash e : s_2$, then $s_2 \sqsubseteq s_1$. Since s_2 can be coerced up to s_1 through subtyping to match the level of the variable, the statement itself can then be typed as $\mathit{cmd}(s_1)$. We shall prove that our type system indeed ensures these properties in Theorems 12-14 below.

Before proceeding, we need to define a way to express that two *states*, i.e. two collections of variable and field environments env_{SV} , are *equal* up to a certain security level s . This relation, written $\Gamma \vdash \mathsf{env}_{SV}^1 =_s \mathsf{env}_{SV}^2$, is given by the rules in Figure 13. Note in particular that the definition implies that env_{SV}^1 and env_{SV}^2 must have the same domain, and this carries over to the inner environments env_F inside env_S . The above definition gives us the following obvious result, which can be shown by induction on the rules of $=_s$:

► **Lemma 9 (Restriction).** *If $\Gamma \vdash \mathsf{env}_{SV}^1 =_s \mathsf{env}_{SV}^2$ and $s' \sqsubseteq s$, then $\Gamma \vdash \mathsf{env}_{SV}^{1'} =_{s'} \mathsf{env}_{SV}^{2'}$.*

Given our annotation of security levels on interfaces as well, we also extend the $=_s$ relation to method tables env_T , and finally to the combined representation of state and code, i.e. env_{ST} .

Next, we need the standard lemmas for strengthening and weakening of the variable environment:

► **Lemma 10 (Strengthening).** *If $\Gamma, x : \mathit{var}(B) \vdash (x, v_1) : \mathsf{env}_V^1 =_s (x, v_2) : \mathsf{env}_V^2$ then also $\Gamma \vdash \mathsf{env}_V^1 =_s \mathsf{env}_V^2$.*

► **Lemma 11 (Weakening).** *If $\Gamma \vdash \mathsf{env}_V^1 =_s \mathsf{env}_V^2$ and $x \notin \mathit{dom}(\mathsf{env}_V^1)$ and $x \notin \mathit{dom}(\mathsf{env}_V^2)$, then also $\Gamma, x : \mathit{var}(B) \vdash (x, v_1) : \mathsf{env}_V^1 =_s (x, v_2) : \mathsf{env}_V^2$ for any B, v, x .*

Both results can be shown by induction on the rules of $=_s$. Furthermore, both of the lemmas can then be directly extended to $\Gamma \vdash \mathsf{env}_{SV}^1 =_s \mathsf{env}_{SV}^2$. With this, we can now state the first of our main theorems:

► **Theorem 12** (Preservation). Assume that $\Gamma \vdash S : cmd(s)$, $\Gamma \vdash env_T$, $\Gamma \vdash env_{SV}$, and $env_T \vdash \langle S, env_{SV} \rangle \rightarrow env'_{SV}$. Then, $\Gamma \vdash env_{SV} =_{s'} env'_{SV}$ for any s' such that $s \not\sqsubseteq s'$.

The Preservation theorem assures us that the promise made by the type $cmd(s)$ is actually fulfilled. If $\Gamma \vdash S : cmd(s)$, then every variable or field written to in S will be of level s or higher; hence every variable or field of a level that is strictly lower than, or incomparable to, s will be unaffected. Thus, the pre- and post-transition states will be equal on all values stored in variables or fields of level s' or lower, since they cannot have been changed during the execution of S . In other words, what is shown to be “preserved” in this theorem is the values at levels lower than, or incomparable to, s .

Note that the theorem does not show preservation of *well-typedness* for the environments (as is otherwise usually required in preservation proofs for type systems). Indeed, a result saying that also $\Gamma \vdash env'_{SV}$ would be pointless. As can be seen in Figure 9, the type judgment $\Gamma \vdash env_{SV}$ only ensures that every field and variable in env_{SV} has *any* type in Γ . The number of declared fields and variables cannot change between the pre- and post-states of a transition (this is ensured by the rule [BS-DECV]); only the stored values can change, but there is no inherent relationship between a value and its assigned security level.

Our next theorem assures us that the type of an expression is also in accordance with the intended meaning, namely: if $\Gamma \vdash e : s$, then every variable (or field) read from in e will be of level s or lower (i.e. no read-down of values from a higher level). We express this by considering two different states, env_{SV}^1 and env_{SV}^2 , which must agree on all values of level s and lower. Evaluating e w.r.t. either of these states should then yield the same result.

► **Theorem 13** (Safety for expressions). Assume that $\Gamma \vdash e : B$ where $B \rightsquigarrow s$, $\Gamma \vdash env_{SV}^1$, $\Gamma \vdash env_{SV}^2$, and $\Gamma \vdash env_{SV}^1 =_s env_{SV}^2$. Then, $env_{SV}^1 \vdash e \rightarrow v$ and $env_{SV}^2 \vdash e \rightarrow v$.

Finally, we can use the preceding two theorems to show soundness for the type system. The soundness theorem expresses that, if a statement S is well-typed to any level s_1 and we execute S with any two states env_{SV}^1 and env_{SV}^2 that agree up to any level s_2 , then the resulting states $env_{SV}^{1'}$ and $env_{SV}^{2'}$ will still agree on all values up to level s_2 . This ensures noninterference, since any difference in values of a higher level than s_2 cannot induce a difference in the computation of values at any lower levels.

► **Theorem 14** (Soundness). Assume that $\Gamma \vdash S : cmd(s_1)$, $\Gamma \vdash env_T$, $\Gamma \vdash env_{SV}^1$, $\Gamma \vdash env_{SV}^2$, $\Gamma \vdash env_{SV}^1 =_{s_2} env_{SV}^2$, $env_T \vdash \langle S, env_{SV}^1 \rangle \rightarrow env_{SV}^{1'}$, and $env_T \vdash \langle S, env_{SV}^2 \rangle \rightarrow env_{SV}^{2'}$. Then, $\Gamma \vdash env_{SV}^{1'} =_{s_2} env_{SV}^{2'}$.

Theorem 14 corresponds to the soundness theorem proved by Volpano, Smith and Irvine [28] for their While-like language. However, given the object-oriented nature of TINY SOL, we can actually take this one step further and allow even parts of the code to vary. Specifically, given two “method table” environments, env_T^1 and env_T^2 , we just require that these two environments agree up to the same level s_2 to ensure agreement of the resulting two states $env_{SV}^{1'}$ and $env_{SV}^{2'}$. We state this in the following theorem:

► **Theorem 15** (Extended soundness). Assume that $\Gamma \vdash S : cmd(s_1)$, $\Gamma \vdash env_T^1$, $\Gamma \vdash env_T^2$, $\Gamma \vdash env_T^1 =_{s_2} env_T^2$, $\Gamma \vdash env_{SV}^1$, $\Gamma \vdash env_{SV}^2$, $\Gamma \vdash env_{SV}^1 =_{s_2} env_{SV}^2$, $env_T^1 \vdash \langle S, env_{SV}^1 \rangle \rightarrow env_{SV}^{1'}$, and $env_T^2 \vdash \langle S, env_{SV}^2 \rangle \rightarrow env_{SV}^{2'}$. Then, $\Gamma \vdash env_{SV}^{1'} =_{s_2} env_{SV}^{2'}$.

4.5 Extending the type system to transactions

A transaction is nothing but a method call with real-valued parameters and `sender` set to an *account* address, which corresponds to a minimal implementation of I^\top . Thus, the theorems from the preceding section can easily be extended to transactions and blockchains.

A blockchain consists of a set of contract declarations DC , followed by a list of transactions \tilde{T} . Hence, we can conclude $\Gamma \vdash DC \tilde{T} : cmd(s)$, if it holds that $\Gamma \vdash DC$ and $\Gamma \vdash \tilde{T} : cmd(s)$. The latter can be simply concluded by the following rules:

$$\begin{array}{c} [\text{T-EMPTY}] \frac{}{\Gamma \vdash \epsilon : cmd(s)} \\ [\text{T-TRANS}] \frac{\Gamma \vdash X.f(\tilde{v}) : n : cmd(s) \quad \Gamma \vdash \tilde{T} : cmd(s)}{\Gamma \vdash A \rightarrow X.f(\tilde{v}) : n, \tilde{T} : cmd(s)} \end{array}$$

This gives us the following two results:

► **Lemma 16.** *If $\Gamma \vdash DC$ and $\langle DC, \text{env}_{ST}^\emptyset \rangle \rightarrow \text{env}_{ST}$, then $\Gamma \vdash \text{env}_{ST}$.*

► **Lemma 17.** *If $\Gamma \vdash A \rightarrow X.f(\tilde{v}) : n, \tilde{T} : cmd(s)$ and $\Gamma \vdash \text{env}_{ST}$ and*

$$\langle A \rightarrow X.f(\tilde{v}) : n, \tilde{T}, \text{env}_{ST} \rangle \rightarrow \langle \tilde{T}, \text{env}_S, \text{env}_T \rangle$$

then also $\Gamma \vdash \text{env}_S, \text{env}_T$ and $\Gamma \vdash \tilde{T}$.

As the initial step (the “genesis event”) does nothing except transforming the declaration DC into the environment representation env_{ST} , the first result is obvious, and as the rule [TRANS] just unwraps a transaction step into a call to the corresponding method, the second result follows directly from the Preservation theorem. This can then be generalised in an obvious way to the whole transaction list. Likewise, the Safety and Soundness theorems can be extended to transactions in the same manner.

4.6 Noninterference and call integrity

As immediately evident from Definition 6 and Theorem 14, well-typedness ensures noninterference:

► **Corollary 18 (Noninterference).** *Assume a set of security levels $\mathcal{S} \stackrel{\Delta}{=} \{L, H\}$, with $L \sqsubseteq L$, $L \sqsubseteq H$ and $H \sqsubseteq H$, and furthermore that $\Gamma \vdash \tilde{T} : cmd(s)$, $\Gamma \vdash \text{env}_{ST}^1$, $\Gamma \vdash \text{env}_{ST}^2$, $\Gamma \vdash \text{env}_{ST}^1 =_L \text{env}_{ST}^2$, $\langle \tilde{T}, \text{env}_{ST}^1 \rangle \rightarrow^* \text{env}_{ST}^{1'}$, and $\langle \tilde{T}, \text{env}_{ST}^2 \rangle \rightarrow^* \text{env}_{ST}^{2'}$. Then, $\Gamma \vdash \text{env}_{ST}^{1'} =_L \text{env}_{ST}^{2'}$,*

From Corollary 18, we then obviously also have that $\Gamma \vdash \text{env}_S^{1'} =_L \text{env}_S^{2'}$, regardless of whether s is L or H . In particular, we can assign security levels to entire contracts, as well as all their members. Thus, our type system can be used to ensure noninterference according to Definition 6.

As we previously argued in Remark 7, noninterference and call integrity are incomparable properties. However, as our next theorem shows, *well-typedness* actually *also* ensures call integrity. This is surprising, so before stating the theorem, we should give some hints as to why this is the case.

The definition of call integrity (Definition 5) requires the execution of any code in a contract C to be unaffected by all contracts in an “untrusted set” \mathcal{Y} , regardless of whether parts of the code in \mathcal{Y} execute before, meanwhile or after the code in C . This is expressed by a quantification over all possible traces resulting from a change in \mathcal{Y} , i.e. either in the code or in the values of the fields. Regardless of any such change, it must hold that the sequence of method calls originating from C be the same.

Noninterference, on the other hand, says nothing about execution traces, but only speaks of the correspondence between values residing in the memory before and after the execution step. The two counter-examples used in Remark 7 made use of this fact:

- The first counter-example had C be unable to perform any method calls at all, thus obviously satisfying call integrity, but allowed different `balance` values to be transferred into it from a “high” context by means of a method call, thereby violating noninterference. However, this situation is ruled out by well-typedness, because well-typedness disallows *any* method calls from a “high” to a “low” context, precisely because every method call may transfer the `value` parameter along with each call.
- The second counter-example had an `if` statement in C (the “low” context) depend on a field value in a “high” context. The two branches then perform two different method calls, thus enabling a change of the “high” context to induce two different execution traces for C . Thus, the example satisfies noninterference, because no value stored in memory is changed, but it obviously does not satisfy call integrity. However, this situation is also ruled out by well-typedness, because the rule [T-IF] does not allow the boolean guard expression e in a “low” context to depend on a value from a “high” context.

Thus, both of the two counter-examples would be rejected by the type system. With a setting of L for the “trusted” segment and H for the “untrusted”,⁵ no values or computations performed in the untrusted segment can affect the values in the trusted segment, *nor* the value of any expression in this segment, nor can it even perform a call into the trusted segment. On the other hand, the trusted segment *can* call out into the untrusted part, but such a call cannot then reenter the trusted segment: it must return before any further calls from the trusted segment can happen.

► **Theorem 19** (Well-typedness implies call integrity). *Let $\mathcal{S} \triangleq \{ L, H \}$ with $L \sqsubseteq L$, $L \sqsubseteq H$ and $H \sqsubseteq H$. Fix the two sets of addresses \mathcal{X} and \mathcal{Y} as in Definition 5, such that $\mathcal{A} = \mathcal{X} \cup \mathcal{Y}$ and $\mathcal{A} = \text{dom}(\text{env}_T)$. Fix a type assignment Γ such that*

- $\forall X \in \mathcal{X} . \Gamma(X) = I_L$ for some I where
 - $\forall p \in \Gamma(I) . \Gamma(I)(p) = \text{var}(B)$ where $B \rightsquigarrow L$, and
 - $\forall f \in \Gamma(I) . \Gamma(I)(f) = \text{proc}(\tilde{B}) : L$ for any \tilde{B}
- and with the level H given to all other interfaces, fields and methods.

Also assume that $\Gamma \vdash T : \text{cmd}(s)$, $\Gamma \vdash \text{env}_{ST}^1$, $\Gamma \vdash \text{env}_{ST}^2$, $\Gamma \vdash \text{env}_{ST}^1 =_L \text{env}_{ST}^2$, $\langle T, \text{env}_{ST}^1 \rangle \xrightarrow{\pi_1} \text{env}_{ST}^{1'}$, and $\langle T, \text{env}_{ST}^2 \rangle \xrightarrow{\pi_2} \text{env}_{ST}^{2'}$. Then, $\pi_1 \downarrow_X = \pi_2 \downarrow_X$, for any $X \in \mathcal{X}$.

Theorem 19 tells us that *every* contract X in the trusted segment \mathcal{X} has call integrity w.r.t. the untrusted segment \mathcal{Y} . This is thus a stronger condition than that of Definition 5, which only defines call integrity for a *single* contract $C \in \mathcal{X}$, rather than for the whole set. This means that our type system will reject cases where e.g. C calls another contract $Z \in \mathcal{X}$ and Z calls `send()` methods of different contracts, depending on a “high” value. As `send()` is always ensured to do nothing, such calls could never lead to C being reentered, so this would actually still be safe, even though Z itself would not satisfy call integrity. Thus, this is an example of what resides in the “slack” of our type system.

However, this situation seems rather contrived, since it depends specifically on the `send()` method, which is always ensured to do nothing except returning. For practical purposes, it would be strange to imagine a contract $C \in \mathcal{X}$ having call integrity w.r.t. \mathcal{Y} , but *without* the other contracts in \mathcal{X} also satisfying call integrity w.r.t. \mathcal{Y} . Thus, our type system seems to yield a reasonable approximation to the property of call integrity.

⁵ This counter-intuitive naming can perhaps best be thought of as indicating our level of *distrust* in a contract.

5 Examples and limitations

Let us see a few examples of the application of the type system. To begin with, consider the first counter-example in Remark 7, which should be ill-typed by the type system. In the counter-example we say that X is Low and Y is High, so we let them both implement the interface $I<S>$ defined as follows:

<pre> 1 interface I<s> { 2 field balance : var(s) 3 method go : proc():s 4 }</pre>	<pre> contract X : I<L> { ... } contract Y : I<H> { ... }</pre>
---	---

where $I<L>$ (resp. $I<H>$) is a shorthand for I_L (resp. I_H) with all occurrences of s within the interface definition replaced by L (resp. H). A part of the failing typing derivation for the body of the method $Y.go()$ in the declaration of contract Y is:

$$\frac{\begin{array}{c} \Gamma(\text{this}) = I<H> \\ \Gamma \vdash \text{this} : I<H> \end{array} \quad \begin{array}{c} \Gamma(I<H>)(\text{balance}) = var(H) \\ \hline \Gamma \vdash \text{this.balance} : var(H) \end{array} \quad \begin{array}{c} \Gamma(X) = I<L> \\ \Gamma \not\vdash X : I<H> \end{array} \quad \begin{array}{c} \Gamma(I<L>)(go) \neq proc() : H \\ \hline \Gamma \not\vdash X.go : proc() : H \end{array}}{\Gamma \not\vdash X.go() : this.balance : cmd(H)} \quad (1)$$

We have that $\Gamma \vdash \text{this.balance} : var(H)$ in contract Y , so in order for the method declaration $go() \{ X.go() : this.balance \}$ in Y to be well-typed, the body of the method must be typable as $proc() : H$ by rule [T-DEC-M]. However, as the derivation in (1) illustrates, this constraint cannot be satisfied, because the lookup $\Gamma(I<L>)(go)$ yields $proc() : L$, but $proc() : H$ is needed, and this cannot be obtained through subtyping, because the $proc(\tilde{B}) : s$ type constructor is contravariant in s .

The above example is simple, since the name X is “hard-coded” directly in the body of $Y.go()$, and therefore the type check fails already while checking the contract definition. However, suppose X were instead received as a parameter. Then the signature of the method $Y.go$ would have to be $\text{method go} : proc(I<H>) : H$ instead, and the type check would then fail at the call-site, if a Low address were passed. The following shows a part of the failing typing derivation for the call $Y.go(X) : this.balance$, where the parameter X is assumed to implement the interface $I<L>$ as before:

$$\frac{\begin{array}{c} \Gamma(X) = I<L> \quad L \sqsubseteq H \quad \frac{\Gamma \vdash L <: H}{\Gamma \vdash var(L) <: var(H)} \quad H \not\sqsubseteq L \\ \Gamma \vdash X : I<L> \end{array} \quad \frac{}{\Gamma \not\vdash I<L> <: I<H>}}{\Gamma \not\vdash Y.go(X) : this.balance : cmd(H)} \quad (2)$$

Here $\Gamma \vdash Y.go : proc(I<H>) : H$ (not shown). The method call expects a parameter of type $I<H>$, but $I<L>$ cannot be coerced up to $I<H>$ through subtyping, because its definition of the method $go()$ has type $proc() : L$, as given in the code listing above, and $\Gamma \not\vdash proc() : L <: proc() : H$ due again to contravariance of the type constructor. Thus we see that the type system indeed prevents calls from High to Low, regardless of whether the Low address is “hard-coded” or passed as a parameter to a High method. However, the aforementioned examples also illustrate a limitation of our type system approach to ensuring call integrity: the entire blockchain must be checked, i.e. both the contracts *and* the transactions. This is necessary since the type check can fail at the call-site of a method, as in the example shown in (2), and the call-site of any method can be a transaction.

```

1 contract X : IBankL {
2   field owner = A;
3   transfer(recipient, amount) {
4     if this.sender = this.owner then
5       recipient
6         .deposit(this.sender):amount
7     else skip
8   }
9   ...
10 }
```

```

contract Y : IBankH {
  field credit = 0;
  deposit(owner) {
    this.credit = this.value;
    this.owner = owner
  }
  ...
}
```

■ **Figure 14** A two-bank setup.

Next, we shall briefly consider two examples, reported by Grishchenko et al. in [13], of Solidity contracts that are misclassified w.r.t. reentrancy by the static analyser Oyente [16]; a false positive and a false negative example.

The false negative example relies on a misplaced update of a field value, just as in the example in Figure 1 (page 2).⁶ In this example, suppose X were assigned the level L and Y the level H. With a transaction $A \rightarrow X.transfer(Y):n$ (for any address A and any amount of currency n), the type system would then correctly reject this blockchain because of the inherent flow from High to Low that is implicit in the call $X.transfer(this)$ issued by Y. The typing derivation would fail in a similar manner as the situation depicted in (2).

The false positive example of Grishchenko et al. from [13] is also similar to the example in Figure 1, but this time just with the assignment to the guard variable correctly placed *before* the method call (i.e. with lines 6 and 7 switched in Figure 1). This too would be rejected by our type system, since it does not take the ordering of statements in sequential composition into account (i.e. rule [T-SEQ]). Thus, this example constitutes a false positive for our type system as well, which is hardly surprising.

Finally, let us consider a true positive example. Figure 14 illustrates a part of the code for two banks, which would allow users to store some of their assets and also to transfer assets between them.⁷ We assume both banks implement the same interface `IBank`, but with different security settings: X is L and Y is H, meaning the latter is untrusted. There is no callback from Y, so in this setup a blockchain with a transaction $A \rightarrow X.transfer(Y, 1):0$ would actually be accepted by the type system, because the Low values from X can safely be coerced up (via subtyping) to match the setting of High on Y.

6 Related work

In light of the visibility and immutability of smart contracts, which makes it hard to correct errors once they are deployed in the wild, it is not surprising that there has been a substantial research effort within the formal methods community on developing formal techniques to

⁶ It also involves the presence of a “default function”, which is a special feature of Solidity. It is a parameterless function that is implicitly invoked by `send()`, thus allowing the recipient to execute code upon reception of a currency transfer. This feature is not present in TINY SOL, yet we can achieve a similar effect by simply allowing the mandatory `send()` method to have an arbitrary method body, rather than just `skip`. This has no effect on the type system and associated proofs, since the `send()` method is treated as any other method therein. Hence, this situation is in principle the same as if the sender had invoked some other method than `send()`, similarly to the example in Figure 1.

⁷ TINY SOL does not have a “mapping” type such as in Solidity, so the setup here is limited to a single user.

prove safety properties of those programs – see, for instance, [26] for a survey. The literature on this topic is already huge and the whole gamut of techniques from the field of verification and validation has been adapted to the smart-contract setting. For example, this includes contributions employing frameworks based on finite-state machines to design and synthesise Ethereum smart contracts [17], a variety of static analysis techniques and accompanying tools, such as those presented in [9, 15, 23, 27], and deductive verification [5, 6, 21], amongst others. The Dafny-based approach reported in [6] is able to model arbitrary reentrancy in a setting with the “gas mechanism”, whereas [4] presents a way to analyse safety properties of smart contracts exhibiting reentrancy in a gas-free setting.

The study in [14] is close in spirit to ours in that it uses a sound type system to guarantee the absence of information flows that violate integrity policies in Solidity smart contracts. That work also presents a type verifier and its prototype implementation within the K-framework [22], which is then applied to analyse more than one hundred smart contracts. However, their technique has not been related to call integrity, which, by contrast, is the focus of our work. Thus, our contribution in the present paper complements this work and serves to further highlight the utility and applicability of secure-flow types in the smart-contract setting. However, there are also clear differences between this aforementioned work and the present one. Most notably, our type system uses a more refined subtyping relation, which also handles subtyping of method and address types, whereas subtyping is not defined for the former in [14], and the latter is not given a type altogether. This gives us a more fine-grained control over the information flow, since it allows us to assign different security levels to a contract and its members. For example, a High contract might have certain Low methods, which hence would not be callable from another High contract, whereas High methods would. This is in line with standard object-oriented principles, e.g. Java-style visibility modifiers.

Another approach to using a type system to ensure smart-contract safety in a Solidity-like language is presented by Crafa et al. in [7]. This work is indeed related to ours in that both are based on well-known typing principles from object-oriented languages, especially subtyping for contract/address types and the inclusion of a “default” supertype for all contracts, similar to our I^\top . However, the aim of [7] is rather different from ours, in that the type system offered in that paper seeks to prevent *runtime errors* that do not stem from a negative account balance, e.g. those resulting from attempts to access nonexistent members of a contract. Incidentally, such runtime errors would *also* be prevented by our type system (rules [T-CALL] and [T-FIELD] in particular), due to our use of “interfaces” as address types, if the converse check (ensuring every declared type in an interface has an implementation) were also performed. However, our focus has been on checking the currency flow, rather than preventing runtime errors of this kind.

The aforementioned paper [7] introduced Featherweight Solidity (FS). Like TINY SOL, FS is a calculus that formalises the core features of Solidity and, as mentioned above, it supports the static analysis of safety properties of smart contracts via type systems. Therefore, the developments in the present paper might conceivably have been carried out in FS instead of TINY SOL. Our rationale for using TINY SOL is that it provided a very simple language that was sufficient to express the property of call integrity, thus allowing us to focus on the core of this property. Of course, “simplicity” is a subjective criterion and the choice of one language instead of another is often a matter of preference and convenience. To our mind, TINY SOL is slightly simpler than FS, which includes functionalities such as callback functions and revert labels. Moreover, the big-step semantics of TINY SOL provided was more convenient for the development of our type system than the small-step semantics given for FS. Furthermore, unlike FS, TINY SOL also formalises the semantics of blockchains. Having said so, TINY SOL and FS are quite similar and it would be interesting to study their similarities

in more detail. To this end, in future work, we intend to carry out a formal comparison of these two core languages and to see which adaptations to our type system are needed when formulated for FS. In particular, we note that FS handles the possibility of an explicit type conversion (type cast) of `address` to `address payable` by augmenting the `address` type with type information about the contract to which it refers. This distinction is not present in our version of TINY SOL, as we require all contracts and accounts to have a default `send()` function, so all addresses are in this sense “payable”. However, our type system does not depend on the presence of a `send()` function, so this difference is not important here.

7 Conclusion and future work

In this paper we studied two security properties, namely call integrity and noninterference, in the setting of TINY SOL, a minimal calculus for Solidity smart contracts. To this end, we rephrased the syntax of TINY SOL to emphasise its object-oriented flavour, gave a new big-step operational semantics for that language and used it to define call integrity and noninterference. Those two properties have some similarities in their definition, in that they both require that some part of a program is not influenced by the other part. However, we showed that the two properties are actually incomparable. Nevertheless, we provided a type system for noninterference and showed that well-typed programs *also* satisfy call integrity. Hence, programs that are accepted by our type systems lie at the intersection between call integrity and noninterference.

A challenging development of our work would be to prove whether the type system exactly characterises the intersection of these two properties, or to find another characterisation of this set of programs. Orthogonally, it would be important to devise type inference algorithms for the present type system, to be used in practical situations where the typing environment is hard to guess. It would also be interesting to compare our typing-based proof method with those proposed, e.g., in [13, 16, 23]. Finally, we also aim at applying our static analysis methodology to many concrete case studies, to better understand the benefits of using a completely static proof technique for call integrity. To do so, it would be useful to extend TINY SOL with a “gas mechanism” allowing one to prove the termination of transactions and to compute their computational cost.

A potential limitation of the approach presented in this paper is that the entire blockchain must be checked to show call integrity of a contract. Indeed, since a typing derivation can fail at the call-site and the call-site of a method can be a transaction, transactions must be well-typed too. In passing, we note that this kind of problem is also present in [25, 28] (and, in general, in many works on type systems for security), where the whole code needs to be typed in order to obtain the desired guarantees. We think that an important avenue for future work, and one we intend to pursue, is to explore whether, and to what extent, other typing disciplines can be employed to mitigate this problem. As mentioned earlier, we also plan to extend the language (and the type system) to enable checking of real-life Solidity contracts; this will also allow us to better assess how (un)feasible it would be to check the whole blockchain.

References

- 1 Luca Aceto, Daniele Gorla, and Stian Lybech. A sound type system for secure currency flow. *CoRR*, abs/2405.12976, 2024. doi:10.48550/arXiv.2405.12976.
- 2 Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *Proc. of POST*, volume 10204 of *LNCS*, pages 164–186. Springer, 2017. doi:10.1007/978-3-662-54455-6_8.

- 3 Massimo Bartoletti, Letterio Galletta, and Maurizio Murgia. A minimal core calculus for solidity contracts. In Cristina Pérez-Solà, Guillermo Navarro-Arribas, Alex Biryukov, and Joaquin Garcia-Alfaro, editors, *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 233–243, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-31500-9_15.
- 4 Christian Bräm, Marco Eilers, Peter Müller, Robin Sierra, and Alexander J. Summers. Rich specifications for Ethereum smart contract verification. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–30, 2021. doi:10.1145/3485523.
- 5 Franck Cassez, Joanne Fuller, and Aditya Asgaonkar. Formal verification of the Ethereum 2.0 Beacon Chain. In *28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 13243 of *LNCS*, pages 167–182. Springer, 2022. doi:10.1007/978-3-030-99524-9_9.
- 6 Franck Cassez, Joanne Fuller, and Horacio Mijail Anton Quiles. Deductive verification of smart contracts with Dafny. In *27th International Conference on Formal Methods for Industrial Critical Systems*, volume 13487 of *LNCS*, pages 50–66. Springer, 2022. doi:10.1007/978-3-031-15008-1_5.
- 7 Silvia Crafa, Matteo Di Pirro, and Elena Zucca. Is solidity solid enough? In *Financial Cryptography Workshops*, 2019.
- 8 The dao smart contract. <http://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413#code>, 2016.
- 9 Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: a static analysis framework for smart contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*, pages 8–15. IEEE / ACM, 2019. doi:10.1109/WETSEB.2019.00008.
- 10 Ethereum Foundation. Solidity documentation. <https://docs.soliditylang.org/>, 2022. Accessed: 2024-01-15.
- 11 Thomas Genet, Thomas P. Jensen, and Justine Sauvage. Termination of Ethereum’s smart contracts. In *Proc. of the 17th International Joint Conference on e-Business and Telecommunications - Volume 2: SECRYPT*, pages 39–51. ScitePress, 2020. doi:10.5220/0009564100390051.
- 12 J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–11, 1982. doi:10.1109/SP.1982.10014.
- 13 Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of ethereum smart contracts. In Lujo Bauer and Ralf Küsters, editors, *Principles of Security and Trust*, pages 243–269, Cham, 2018. Springer International Publishing.
- 14 Xinwen Hu, Yi Zhuang, Shangwei Lin, Fuyuan Zhang, Shuanglong Kan, and Zining Cao. A security type verifier for smart contracts. *Comput. Secur.*, 108:102343, 2021. doi:10.1016/j.cose.2021.102343.
- 15 Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. ZEUS: analyzing safety of smart contracts. In *25th Annual Network and Distributed System Security Symposium*. The Internet Society, 2018. URL: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_09-1_Kalra_paper.pdf.
- 16 Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proc. SIGSAC Conf. on Computer and Communications Security*, pages 254–269. ACM, 2016. doi:10.1145/2976749.2978309.
- 17 Anastasia Mavridou and Aron Laszka. Designing secure Ethereum smart contracts: A finite state machine based approach. In *22nd Conference on Financial Cryptography and Data Security*, volume 10957 of *LNCS*, pages 523–540. Springer, 2018. doi:10.1007/978-3-662-58387-6_28.
- 18 Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer*. Springer-Verlag London, 2007. doi:10.1007/978-1-84628-692-6.
- 19 The parity wallet breach. <https://www.coindesk.com/30-million-ether-reported-stolen-parity-wallet-breach/>, 2017.

- 20 The parity wallet vulnerability. <https://paritytech.io/blog/security-alert.html>, 2017.
- 21 Daejun Park, Yi Zhang, and Grigore Rosu. End-to-end formal verification of Ethereum 2.0 Deposit Smart Contract. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV Proceedings, Part I*, volume 12224 of *Lecture Notes in Computer Science*, pages 151–164. Springer, 2020. doi:10.1007/978-3-030-53288-8_8.
- 22 Grigore Rosu and Traian-Florin Serbanuta. An overview of the K semantic framework. *J. Log. Algebraic Methods Program.*, 79(6):397–434, 2010. doi:10.1016/j.jlap.2010.03.012.
- 23 Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. ethor: Practical and provably sound static analysis of ethereum smart contracts. In *Proc. of SIGSAC Conf. on Computer and Communications Security*, pages 621–640. ACM, 2020. doi:10.1145/3372297.3417250.
- 24 Pablo Lamela Seijas, Simon J. Thompson, and Darryl McAdams. Scripting smart contracts for distributed ledger technology. *IACR Cryptol. ePrint Arch.*, 2016:1156, 2016.
- 25 Geoffrey Smith and Dennis M. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. of 25th POPL*, pages 355–364. ACM, 1998.
- 26 Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. A survey of smart contract formal specification and verification. *ACM Computing Surveys (CSUR)*, 54(7):148:1–148:38, 2020. doi:10.1145/3464421.
- 27 Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. Securify: Practical security analysis of smart contracts. In *Proc. of SIGSAC Conference on Computer and Communications Security*, pages 67–82. ACM, 2018. doi:10.1145/3243734.3243780.
- 28 Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4, August 2000. doi:10.3233/JCS-1996-42-304.
- 29 Renlord Yang, Toby Murray, Paul Rimba, and Udaya Parampalli. Empirically analyzing Ethereum’s gas mechanism. In *Proc. of IEEE European Symposium on Security and Privacy Workshops*, pages 310–319. IEEE, 2019. doi:10.1109/EuroSPW.2019.00041.

Runtime Instrumentation for Reactive Components

Luca Aceto  

Reykjavik University, Iceland

Gran Sasso Science Institute, L'Aquila, Italy

Duncan Paul Attard  

University of Glasgow, UK

Adrian Francalanza  

University of Malta, Msida, Malta

Anna Ingólfssdóttir  

Reykjavik University, Iceland

Abstract

Reactive software calls for instrumentation methods that uphold the reactive attributes of systems. Runtime verification imposes another demand on the instrumentation, namely that the trace event sequences it reports to monitors are *sound* – that is, they reflect actual executions of the system under scrutiny. This paper presents RIARC, a novel decentralised instrumentation algorithm for outline monitors meeting these two demands. Asynchrony in reactive software complicates the instrumentation due to potential trace event loss or reordering. RIARC overcomes these challenges using a next-hop IP routing approach to rearrange and report events soundly to monitors.

RIARC is validated in two ways. We subject its corresponding implementation to rigorous systematic testing to confirm its correctness. In addition, we assess this implementation via extensive empirical experiments, subjecting it to large realistic workloads to ascertain its reactivity. Our results show that RIARC optimises its memory and scheduler usage to maintain latency feasible for soft real-time applications. We also compare RIARC to inline and centralised monitoring, revealing that it induces comparable latency to inline monitoring in moderate concurrency settings where software performs long-running, computationally-intensive tasks, such as in Big Data stream processing.

2012 ACM Subject Classification Software and its engineering → Software verification and validation

Keywords and phrases Runtime instrumentation, decentralised monitoring, reactive systems

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.2

Related Version *Extended Version:* <https://arxiv.org/abs/2406.19904> [8]

Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact):*

<https://doi.org/10.4230/DARTS.10.2.1>

Software (Source Code): <https://doi.org/10.5281/zenodo.10634182> [15]

Funding This work is supported by the Reykjavik University Research Fund, the Doctoral Student Grant (No: 207055) and the MoVeMnt project (No: 217987) under the IRF, and the STARDUST project (No: EP/T014628/1) under the EPSRC.

Acknowledgements We thank our reviewers and the Artefact Evaluation Committee for their feedback. Thanks also to Keith Bugeja, Simon Fowler, Simon Gay, and Phil Trinder for their input.

1 Introduction

Modern software is generally built in terms of concurrent components that execute without relying on a global clock or shared state [87]. Instead, components interact via non-blocking messaging, creating a loosely-coupled architecture known as a *reactive system* [9, 94], which

- responds in a timely manner (is *responsive*),
- remains available in the face of failure (is *resilient*),



© Luca Aceto, Duncan Paul Attard, Adrian Francalanza, and Anna Ingólfssdóttir;

licensed under Creative Commons License CC-BY 4.0

38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 2; pp. 2:1–2:33



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

- reacts to inputs from users or their environment (is *message-driven*), and
- grows and shrinks to accommodate varying computational loads (is *elastic*).

The real-world behaviour of reactive systems is hard to understand statically, and *monitoring* is used to inspect their operation at *runtime*, e.g. for debugging [111], security checking [62], profiling [76], resource usage analysis [36], etc. This paper considers runtime verification (RV), an application of monitoring used to detect whether the *current* execution of a system under scrutiny (SuS) deviates from its correct behaviour [17, 71, 22]. A RV monitor is a *sequence recogniser* [123, 101]: a state machine that incrementally analyses a *finite* fragment of the runtime information exhibited by a SuS to reach an *irrevocable* verdict (see [6, 5] for details).

Instrumentation lies at the core of runtime monitoring [70, 22, 64]. It is the mechanism by which runtime information from a SuS is extracted and reported to monitors as a stream of system events called a *trace*. Software is typically instrumented in one of two ways. Inline instrumentation, or *inlining*, modifies the SuS by injecting tracing instructions at specific joinpoints, e.g. using AspectJ [90] or BCEL [54]. Outline instrumentation, or *outlining*, uses an external tracing infrastructure to gather events, e.g. LTTng [56] or OpenJ9 [58], thereby treating the SuS as a *black box*. A key requirement setting RV apart from monitoring, e.g., telemetry [85] or profiling [121, 26], is that the instrumentation must report *sound traces*.

- **Definition 1 (Sound traces).** A finite trace T is sound w.r.t. a system component P iff it is
1. Complete. T contains all the events exhibited by P so far, and
 2. Consistent. The event sequence in T reflects the order the events occur locally at P . \square

Traces violating this soundness invariant are unfit for RV, as omitted, spurious, or out-of-sequence events incorrectly characterise the system behaviour, *nullifying* the verdicts that monitors flag [22, 52]. Reactive software imposes another requirement: that the instrumentation *safeguards* the responsive, resilient, message-driven, and elastic attributes of the SuS. This necessitates an instrumentation method which is itself *reactive*, in order to

1. not hamper the SuS by inducing unfeasible runtime overhead (is responsive),
2. permit monitors to fail independently of SuS components (is resilient),
3. react to trace events without blocking the SuS (is message-driven), and
4. grow and shrink in proportion to the size of the SuS (is elastic).

Limitations of current RV instrumentation methods. State-of-the-art RV tools use instrumentation methods that do not satisfy *all* of the conditions 1–4 above. This renders them inapplicable to reactive software; see [64, tables 3 and 4] for details. Many approaches, including [24, 31, 49, 75, 110, 122, 127, 19], assume systems with a *fixed* architecture where the number of components remains constant at runtime, failing to meet condition 4. Works foregoing the assumption of a fixed system size, such as [45, 91, 60, 59, 25, 31, 68, 3], inline the SuS with monitors *statically*. Inlining monitors pre-deployment inherently accommodates systems that grow and shrink (condition 4) as a by-product of the embedded monitor code that executes on the same thread of system components; see fig. 1a. This scheme, however, has shortcomings that make it less suited to reactive software. Recent studies [22, 52] observe that the lock-step execution of the SuS and monitors can impair the operation of the instrumented system, e.g. slow runtime analyses manifest as high latencies [37], and faulty monitors may break the system [69], which do not meet conditions 1 and 2 (e.g. M_Q in fig. 1a). Other works [46, 16] argue that errors, such as deadlocks or component crashes, are hard to detect since the monitoring logic shares the runtime thread of the affected component. Entwining the execution of the SuS and monitors may also diminish the scalability, performance, and resource usage efficiency of the monitored system because inlined monitor code cannot be run on separate threads [12]. Lastly, inlining is *incompatible* with unmodifiable software, such as closed-source components (e.g. R in figs. 1a–1c), making outlining the only alternative.

Outline instrumentation *can* address the limitations of inlining by isolating the SuS and its monitors (works [45, 37, 38] that view externalised monitors as “outline” embed tracing code to extract events from the SuS, subjecting them to the cons of inlining). The latest survey on decentralised RV [71, tables 1 and 2] establishes that outlining-based tools, e.g. [50, 18, 19, 72, 37, 38, 125, 65], are variations on *centralised* instrumentation. In this set-up, events exhibited by SuS components are funnelled through a *global* trace buffer (e.g. $\kappa_{\{P,Q,R\}}$ in fig. 1b) that a singleton monitor can analyse asynchronously, meeting condition 3. Yet, the central buffer introduces contention and sacrifices the scalability of the SuS [11], violating condition 4. Centralised architectures are prone to single point of failures (SPOFs) [94, 93] (violating condition 2), which is not ideal for monitoring medium-scale reactive systems.

Contribution. We propose RIARC, a *decentralised* instrumentation algorithm for outline monitors that overcomes the above shortcomings, fulfilling conditions 1–4. Outline monitors minimise latency effects due to slow trace event analyses associated with inlining (meeting condition 1). While RIARC does not handle monitor failure explicitly, it intrinsically enjoys a degree of fault tolerance by isolating the SuS and its decentralised monitor components (meeting condition 2); e.g. monitors $M_{\{P\}}$ and $M_{\{Q,R\}}$ in fig. 1c. RIARC uses a tracing infrastructure to obtain system events passively without modifying the SuS (meeting condition 3). The algorithm equips each isolated monitor with a *local* trace buffer, using it to report events based on the SuS components a monitor is tasked to analyse (e.g. buffers $\kappa_{\{P\}}$ and $\kappa_{\{Q,R\}}$ in fig. 1c). RIARC reorganises its instrumentation set-up to reflect dynamic changes in the SuS. It reacts to specific events in traces to instrument monitors for new SuS components and to remove redundant monitors when it detects graceful or abnormal component terminations. This enables RIARC to grow and shrink the verification set-up on demand (meeting condition 4). Given the challenges of fulfilling the conditions 1–4, we scope our work to settings where communication is reliable (i.e., no message corruption, duplication, and loss) [57] and Byzantine failures do not arise [96].

To the best of our knowledge, the approach RIARC advocates is novel. One reason why outlining has never been adopted for decentralising monitors are the onerous conditions 1–4 imposed by reactive software. Utilising non-invasive tracing makes our set-up necessarily *asynchronous*. At the same time, this complicates the instrumentation, which must ensure trace soundness (def. 1), notwithstanding the inherent phenomena arising from the concurrent execution of the SuS and monitors, e.g. trace event reordering and process crashes. Consequently, the second reason is that the overhead incurred to uphold this invariant – in addition to scaling the verification set-up as the SuS executes – is perceived as prohibitive when compared to inlining. This opinion is often reinforced when the viability of outline instrumentation is predicated on empirical criteria tied to monolithic, batch-style programs, that *may not* apply to reactive software (e.g. percentage slowdown); e.g. see [97, 114, 113, 47, 46, 119, 30, 98].

This paper shows how instrumenting outline monitors under conditions 1–4 can be achieved using a decentralised approach that guarantees def. 1, while *also* exhibiting overheads considered feasible for typical soft real-time reactive systems. Concretely, we

- (i) recall the benefits of the actor model [82, 10] for building reactive systems and argue how our model of processes and tracers readily maps to that setting, sec. 2;
- (ii) give a decentralised instrumentation algorithm for outline monitors, detailing how the reactive characteristics of the SuS can be preserved whilst ensuring def. 1, sec. 3;
- (iii) show the implementability of our algorithm in an actor language and systematically validate the correctness of its corresponding implementation w.r.t. def. 1 by exhaustively inducing interleaved executions for a selection of instrumented systems, sec. 4;

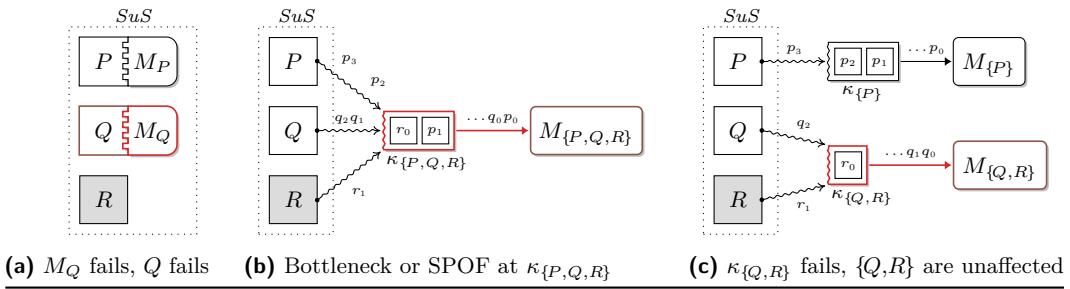


Figure 1 P, Q, R instrumented in inline (*left*), centralised (*middle*) and decentralised (*right*) modes.

- (iv) back up the feasibility of the implemented algorithm via a comprehensive empirical study that uses various workload configurations surpassing the state of the art, showing that the induced overhead minimally impacts the reactive attributes of the SuS, sec. 5.

The extended version [8] contains the full details about RIARC and further discussion of our experiments and results. That material is ancillary to the one presented in this paper.

2 A computational model for reactive systems

The actor model [82, 10] emerged as *the* paradigm to design and build reactive systems [33]. *Actors* – the units of decomposition in this model – are abstractions of concurrent entities that share no mutable memory with other actors. Instead, actors interact through asynchronous message passing and alter their internal state based on the messages they consume. Asynchronous communication decouples actors spatially and temporally, which fully isolates system components and establishes the foundation for resiliency and elasticity [32, 94]. Each actor is equipped with an incoming message buffer called the *mailbox*, from which messages deposited by other actors can be selectively read. Besides sending and receiving messages, actors can *spawn* other actors. Actors in a system are addressable by their unique process identifier (PID), which they use to engage in directed, *point-to-point* communication. This idea of addressability is central to the actor model: it enables reasoning about decentralised computation, as the identity of components or messages can be propagated through a system and used in handling complex tasks, such as process registration and failure recovery [33]. As is often the case in decentralised computations, we assume that messages exchanged between pairs of processes are always received in the order in which they have been sent [43].

Frameworks, notably Erlang [12], Elixir [88], Akka [1] for Scala [117], along with others [118, 130], instantiate the actor model. We adopt Erlang since its ecosystem is specifically engineered for highly-concurrent, soft real-time reactive systems [131, 13, 44]. The Erlang virtual machine (EVM) implements actors as lightweight processes. It employs *per process* garbage collection that, unlike the JVM, does not subject the virtual machine to global unpredictable pauses [86, 116]. This factor minimises the impact on the soft real-time properties of a system *and* is also crucial to the empirical evaluation of sec. 5 since it stabilises the variance in our experiments. The EVM exposes a flexible *process tracing* API aimed at reactive software [42]. Erlang provides other components, e.g. supervision trees, message queues, etc., for building fault-tolerant distributed applications. While we scope our work to fault-free settings (see sec. 1), adopting Erlang gives us the foundation upon which our work can be naturally extended to address these aspects. Henceforth, we follow the established convention in Erlang literature and use the terms *actor*, *process*, and *component* synonymously.

2.1 Process tracing and trace partitioning

Processes in a concurrent system form a *tree*, starting at the *root* process that spawns *child* processes, and so forth¹. Concurrency induces inherent *partitions* to the execution of the SuS in the form of isolated traces that reflect the *local* behaviour at each process [19]. RIARC exploits this aspect to attain several benefits. First, one can *selectively* specify the SuS processes to be instrumented. The upshot is that fewer trace events need to be gathered, improving *efficiency*. Another benefit of partitioned traces is that each process can be dynamically instrumented, free from assumptions about the number of processes the SuS is expected to have. This makes the RV set-up *elastic*. Lastly, the instrumentation set-up can *partially fail*, as faulty SuS or monitor processes do not imperil the execution of one another.

► **Example 2** (Trace partitions). Trace partitions enable RIARC to instrument a system in various arrangements. Fig. 2a depicts an interaction sequence for the execution of the SuS from sec. 1. In this interaction, the root process, P , spawns Q and communicates with it, at which point Q spawns process R ; P and Q eventually terminate. We denote the process *spawning* and *termination* trace events by \diamond and \star , and use $!$ and $?$ for *send* and *receive* events respectively. The *sound* trace partitions for the processes in fig. 2a are “ $\neg \diamond_P . !_P . \star_P$ ” for P , “ $?_Q . \neg \diamond_Q . \star_Q$ ” for Q , and the empty trace for R . □

A centralised set-up such as that of fig. 1b can be obtained by instrumenting $\{P, Q, R\}$ with one monitor, $M_{\{P, Q, R\}}$, whereas instrumenting the components $\{P\}$ and $\{Q, R\}$ with monitors $M_{\{P\}}$ and $M_{\{Q, R\}}$ gives the decentralised arrangement of fig. 1c. Each of these instrumentation arrangements generates different executions.

► **Example 3** (Sound traces). For the case of fig. 1b, RIARC can report to $M_{\{P, Q, R\}}$ one of four possible traces “ $\neg \diamond_P . !_P . \star_P . ?_Q . \neg \diamond_Q . \star_Q$ ”, “ $\neg \diamond_P . !_P . ?_Q . \star_P . \neg \diamond_Q . \star_Q$ ”, “ $\neg \diamond_P . !_P . ?_Q . \neg \diamond_Q . \star_P . \star_Q$ ”, or “ $\neg \diamond_P . !_P . ?_Q . \neg \diamond_Q . \star_Q . \star_P$ ”. These *sound* traces result from the interleaved execution of processes P , Q . Any other trace, e.g. “ $\neg \diamond_P . \star_P . ?_Q . \neg \diamond_Q . \star_Q$ ” or “ $\neg \diamond_P . !_P . \star_P . ?_Q . \star_Q . \neg \diamond_Q$ ”, is *unsound* since it contradicts the local behaviour at processes P and Q of fig. 2a. The former trace omits the request $!_P$ that P makes to Q (it is *incomplete* w.r.t. P), and the latter trace inverts $\neg \diamond_Q$ and \star_Q , suggesting that Q spawns R after Q terminates (it is *inconsistent* w.r.t. Q). □

► **Example 4** (Separate instrumentation). Fig. 2b shows another decentralised set-up, where P , Q , and R are instrumented separately. In this case, the instrumentation should report to $M_{\{P\}}$, $M_{\{Q\}}$ and $M_{\{R\}}$ the events observed *locally* at each process, as stated in ex. 2. □

RIARC makes two assumptions about process tracing in order to support the instrumentation arrangements shown in figs. 1b, 1c, and 2b:

A₁ *Tracing processes sets*. Tracing can gather events for *sets* of SuS processes, e.g. $\kappa_{\{P, Q, R\}}$ in fig. 1b gathers the events of $\{P, Q, R\}$, and $\kappa_{\{Q, R\}}$ in fig. 1c gathers the events of $\{Q, R\}$.

A₂ *Tracing inheritance*. Tracing gathers the events of a SuS process *and* the children it spawns by default to eliminate the risk that trace events from child processes are missed.

We opt for tracing inheritance since it follows established centralised RV monitoring tools, including [18, 41, 50, 110]. In fact, tracing assumptions A₁ and A₂ mean that centralised set-ups like that of fig. 1b can be obtained just by tracing the root process P . Tracing inheritance requires the instrumentation to *intervene* if it needs to channel the events of a child process into a *new* trace partition that is *independent* from that of its parent, e.g. as in

¹ For example, using `spawn()` in Erlang [42] and Elixir [88], `ActorContext.spawn()` in Akka [1], `Actor.createActor()` in Thespian [118], `CreateProcess()` in Windows [108], etc.

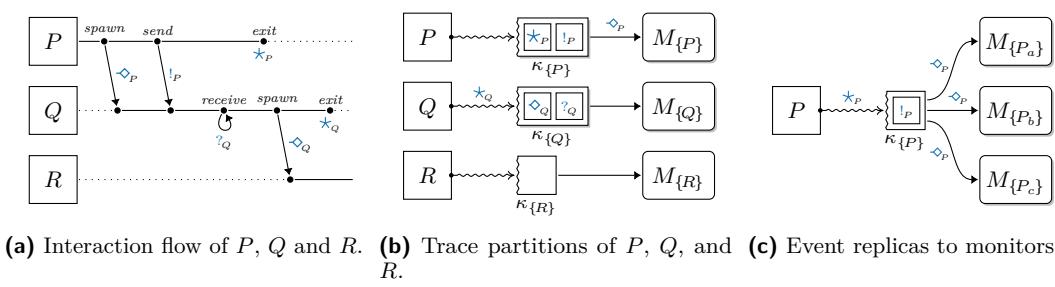


Figure 2 SuS with processes P , Q , and R instrumented with independent monitors.

fig. 1c. In such cases, the instrumentation must first stop tracing the child process, allocate a fresh trace buffer, and resume tracing the child process. The out-of-sync execution of the SuS and instrumentation complicates the creation of these new trace partitions because it can lead to reordered or missed events. This, in turn, would violate trace soundness, def. 1.

We supplement A₁ and A₂ with the following to keep our exposition in sec. 3 manageable:

- A₃** *Single-process tracing.* Any SuS process can be traced *at most* once at any point in time.
- A₄** *Causally-ordered spawn events.* Tracing gathers the spawn trace event of a parent process before *all* the events of the child process spawned by that parent, e.g. if P spawns Q , and Q receives, as in fig. 2a, the reported sequence is “ $\diamond_P \cdot ?_Q$ ” rather than “ $?_Q \cdot \diamond_P$ ”.

The constraint of tracing assumption A₃ is easily overcome by replicating trace events for a process and reporting them to different monitors (e.g. the events in the trace partition of process P are replicated to monitors $M_{\{P_a\}}$, $M_{\{P_b\}}$, $M_{\{P_c\}}$ in fig. 2c). Tracing assumption A₄ requires trace buffers to reorder \diamond events using the spawner and spawned process information carried by each event before reporting them to monitors. Sec. 3.3 gives more details.

► **Example 5 (Unsound traces).** Fig. 3a shows one possible configuration that can be reached by our three-process system introduced in fig. 2a, where the trace buffer $\kappa_{\{P\}}$ contains the events for both P and Q . The trace in buffer $\kappa_{\{Q\}}$ is unsound, as it inaccurately characterises the local behaviour of process Q (the sound trace for Q should be “ $?_Q \cdot \diamond_Q \cdot \star_Q$ ”, not “ \star_Q ”). □

RIARC programs trace buffers to coordinate with one another to ensure that sound traces are invariably reported to monitors. We refer to a trace buffer and the coordination logic it encapsulates as a *tracer*. RIARC employs an approach based on *next-hop routing* in IP networks [80, 104] to counteract the effects of trace event reordering and loss by rearranging and forwarding events to different tracers. Fig. 3b conveys our organisation of tracers (refer to [8, fig. 10 in app. A] for legend). Sec. 3 details how RIARC dynamically reorganises the tracer choreography and performs next-hop routing.

2.2 Modelling decentralised instrumentation

Since RV monitors are passive verdict-flagging machines (refer to sec. 1), they are orthogonal to our instrumentation. We, thus, focus our narrative on tracers and omit monitors, except when relevant in the surrounding context. The model assumes a set of SuS process, $P, Q, R \in \text{PRC}$, and tracer names, $T \in \text{TRC}$, together with a countable set of PID values to reference processes. We distinguish between SuS and tracer PIDs, which we denote respectively by the sets, $p_s, q_s \in \text{PID}_S$ and $p_T, q_T \in \text{PID}_T$. The variables i_s and j_s , and i_T and j_T range over PIDs from the corresponding sets PID_S and PID_T . We also assume the function signature sets, $f_S \in \text{SIG}_S$, $f_T \in \text{SIG}_T$, and, $f_M \in \text{SIG}_M$, to denote SuS, tracer, and RV monitor functions, together with

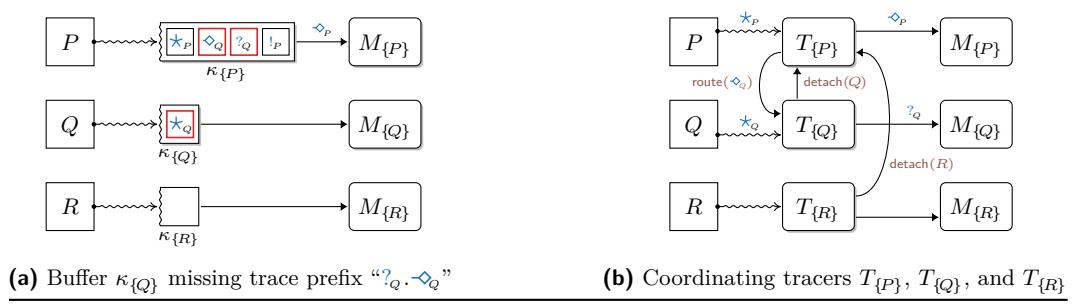


Figure 3 Choreographed tracers coordinating to ensure sound traces.

the variables ς_S , ς_T , and ς_M that range over each signature set. New SuS processes are created via the function $\text{spwn}(\varsigma_S)$ that accepts the function signature ς_S to be spawned, and returns a fresh PID, ι_S . We overload spwn to spawn tracer signatures ς_T equivalently, returning corresponding PIDs, ι_T . The function self obtains the PID of the process invoking it. We write P as shorthand for a singleton process set $\{P\}$ to simplify notation.

RIARC uses three message types, $\tau \in \{\text{evt}, \text{dtc}, \text{rtd}\}$. These determine when to *create* or *terminate* tracer processes, and what trace events to *route* between tracers:

- **evt** are *trace events* gathered via process tracing,
- **dtc** are *detach requests* that tracers exchange to reorganise the tracer choreography, and
- **rtd** are *routing packets* that transport **evt** or **dtc** messages forwarded between tracers.

We encode messages m as tuples. Trace event messages, $\langle \text{evt}, \ell, \iota_S, j_S, \varsigma_S \rangle$, comprise the event label ℓ that ranges over the SuS events \diamond (*spawn*), \star (*exit*), $!$ (*send*), and $?$ (*receive*). The PID value ι_S identifies the SuS process exhibiting the trace event, and is defined for *all* events. The SuS PID j_S and function signature ς_S depend on the type of the event. Tbl. 1a catalogues the values defined for each event. We write trace events in their shorthand form, omitting undefined values (denoted by \perp), e.g. $\langle \text{evt}, \star, \iota_S \rangle$ instead of $\langle \text{evt}, \star, \iota_S, \perp, \perp \rangle$.

Table 1 Trace event (**evt**), detach request (**dtc**), and routing packet (**rtd**) message index names.

(a) Messages encoding *spawn*, *exit*, *send*, and *receive* events.

Label ℓ	Index	Description (ι_S and j_S are SuS PIDs)
\diamond	$e.\iota_S$	Parent PID spawning new child PID j_S
\diamond	$e.j_S$	Child PID spawned by parent PID ι_S
\diamond	$e.\varsigma_S$	Signature ς_S spawned by parent PID ι_S
\star	$e.\iota_S$	Terminated PID
\star	$e.j_S, e.\varsigma_S$	Undefined for exit events
$!$	$e.\iota_S$	Sending PID
$!$	$e.j_S$	Recipient PID
$!$	$e.\varsigma_S$	Undefined for send events
$?$	$e.\iota_S$	Recipient PID
$?$	$e.j_S, e.\varsigma_S$	Undefined for receive events

(b) Detach and routing messages.

Index	Description
$m.\tau$	Message type: event (evt) detach (dtc), routing (rtd)
$d.\iota_T$	PID of tracer requesting detach of SuS PID ι_S
$d.\iota_S$	PID of SuS process to stop tracing
$r.\iota_T$	PID of tracer that starts routing message m
$r.m$	Embedded evt or dtc message being routed

■ **Table 2** RIARC approach to ensure trace soundness (def. 1) and reactive instrumentation (sec. 1).

Requirement	Approach
R ₁ Growing the set-up	Instrument tracers on-demand to create new trace partitions
R ₂ Ensuring complete traces	Route trace events to deliver them to the correct tracer
R ₃ Ensuring consistent traces	Prioritise routed trace events before others
R ₄ Isolating tracers	Detach tracers from others once all trace events are routed
R ₅ Minimising overhead	Target specific processes to instrument
R ₆ Shrinking the set-up	Garbage collect redundant tracers and monitors

Detach request messages have the form $\langle \text{dtc}, \iota_T, \iota_S \rangle$. A tracer with the PID ι_T uses **dtc** to request that another tracer *stop* tracing the SuS PID ι_S . Routing packet messages, $\langle \text{rtd}, \iota_T, m \rangle$, move **evt** and **dtc** messages between tracers. The PID ι_T identifies the tracer that embeds the message m into the routing packet and dispatches it to other tracers. Tbl. 1b summarises detach request and routing packet messages.

► **Note 6 (Notation).** We reserve the variables e , d , and r for the messages types **evt**, **dtc**, and **rtd** respectively. Our model uses the suggestive dot notation $(.)$ to index message fields, e.g. $m.\tau$ reads the message type, $e.\ell$ reads the trace event label, etc. (see tbl. 1). For simplicity, we occasionally write the label ℓ in lieu of the full trace event form, e.g. we write \star instead of $\langle \text{evt}, \star, \iota_S \rangle$, etc. □

3 Decentralised instrumentation

Our reason for encapsulating trace buffers and their coordination logic as tracers stems from the actor model. Trace buffers align with actor mailboxes, which localise the tracer state and enable tracers to run *independently*. The main logic replicated at each tracer is given in algs. 1–3. Tracers operate in two modes, *direct* (\circ) and *priority* (\bullet), to counteract the effects of trace event reordering. We organise our tracer logic in algs. 1 and 3 to reflect these modes, respectively. Algs. 1 and 3 use the function **ANALYSE.EVT**, which analyses events; see [8, app. C.5.2] for details. Auxiliary tracer logic referenced in this section is given in [8, app. A].

Every tracer maintains an internal state σ consisting of the following three maps:

- the *routing* map, Π , governing how events are routed to other tracers,
 - the *instrumentation* map, Λ , that determines which SuS processes to instrument, and
 - the *traced-processes* map, Γ , tracking the SuS process set that the tracer currently traces.
- Tbl. 2 summarises the challenges that RIARC needs to overcome to attain the reactive characteristics stated in sec. 1. Requirements R₁ and R₆ in tbl. 2 oblige the instrumentation to reorganise dynamically while the SuS executes to preserve its *elasticity*. Requirement R₄ offers a modicum of *resiliency* between the SuS and tracer processes, whereas R₅ minimises the instrumentation overhead by gathering only the events monitors require. This keeps the overall set-up *responsive*. Since RIARC builds on the actor model, it fulfils the *message-driven* requirement intrinsically. *Trace soundness* is safeguarded by requirements R₂ and R₃.

The operations **TRACE**, **CLEAR** and **PREEMPT** give access to the tracing infrastructure. **TRACE**(ι_S, ι_T) enables a tracer with PID ι_T to register its interest in receiving trace events of a SuS process with PID ι_S . This operation can be undone using **CLEAR**(ι_S, ι_T), which *blocks* the calling tracer ι_T and returns once all the trace event messages for the SuS process ι_S that are in transit to the tracer ι_T have been delivered to ι_T . It is worth remarking that this behaviour conforms to our proviso in sec. 1, i.e., no communication faults. **PREEMPT**(ι_S, ι_T) combines

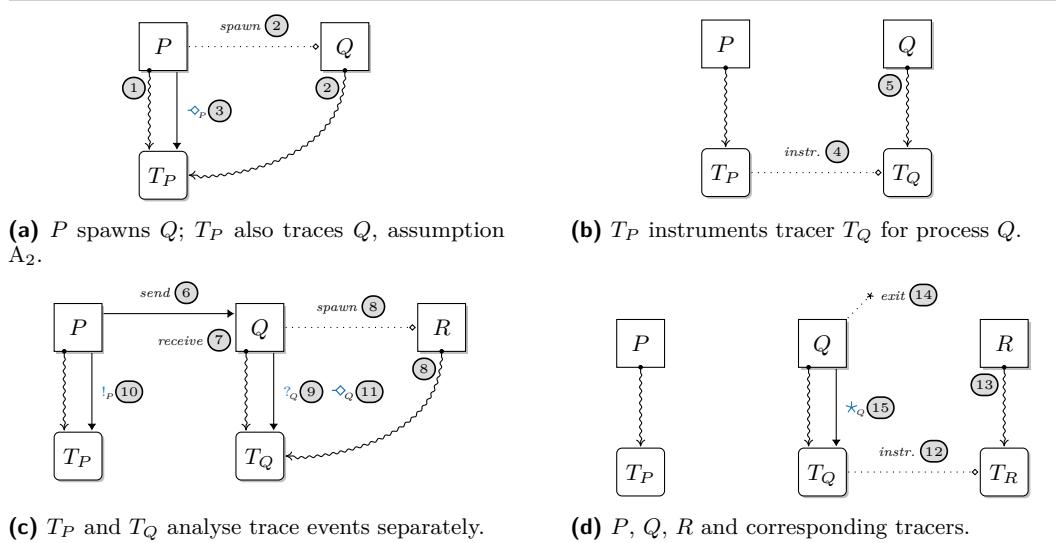
CLEAR and TRACE. It enables the tracer pre-empting ι_T to take control of tracing the SuS process ι_s from another tracer ι'_T that is currently tracing ι_s . Tracers use CLEAR or PREEMPT to modify the default process-tracing inheritance behaviour that tracing assumption A₂ describes. We refer readers to [8, alg. 5 in app. A] for the specifics of these operations.

We focus our presentation in secs. 3.1–3.6 of how RIARC addresses the challenges listed in tbl. 2 on the set-up of fig. 2b, where the processes P , Q and R , are instrumented separately. This specific case highlights two aspects. First, it *emphasises* the complications that RIARC overcomes to establish the desired set-up while ensuring trace soundness, def. 1. Second, fig. 2b *covers all* other possible instrumentation set-ups. Disjoint sets of SuS processes, including the one shown in fig. 1c, can be obtained when tracers do not act on certain \diamond (*spawn*) events, as sec. 3.1 explains. Notably, *any* centralised set-up, e.g. the one in fig. 1b, emerges naturally when the root tracer disregards all \diamond events exhibited by the SuS.

► **Note 7 (Naming conventions).** For clarity, we adopt the convention that a SuS process P is spawned from the signature f_{s_P} and is assigned the PID p_s . A tracer for P is named T_P (short for $T_{\{P\}}$) and has the PID p_T . Other processes are treated likewise, e.g. the SuS process Q has signature f_{s_Q} , PID q_s , while the tracer T_Q for Q has PID q_T , etc. □

3.1 Growing the set-up

Fig. 4 illustrates how the hierarchical creation sequence of SuS processes described in sec. 2.1 is exploited to instrument separate tracers. RIARC programs tracers to react to \diamond (*spawn*) events in the trace. In fig. 4a, the root tracer T_P traces process P , step ①. When P spawns process Q , Q automatically inherits T_P (tracing assumption A₂ from sec. 2.1). Steps ② in fig. 4a emphasise that tracing inheritance is instantaneous. The event $e = \langle \text{evt}, \diamond, p_s, q_s, f_{s_Q} \rangle$ is generated by P when it spawns its child Q , step ③ in fig. 4a. The PID values of the parent and child processes carried by e , namely p_s and q_s , are accessed via the indexes $e.\iota_s$ and $e.\jmath_s$ respectively (see tbl. 1a). Tracer T_P uses this PID information to instrument a new tracer T_Q for process Q in step ④ of fig. 4b. By invoking PREEMPT(q_s, q_T), T_Q takes over tracing process Q from the former tracer T_P going forward. T_Q creates a new trace partition for



■ **Figure 4** Growing the tracer instrumentation set-up for processes P , Q and R (monitors omitted).

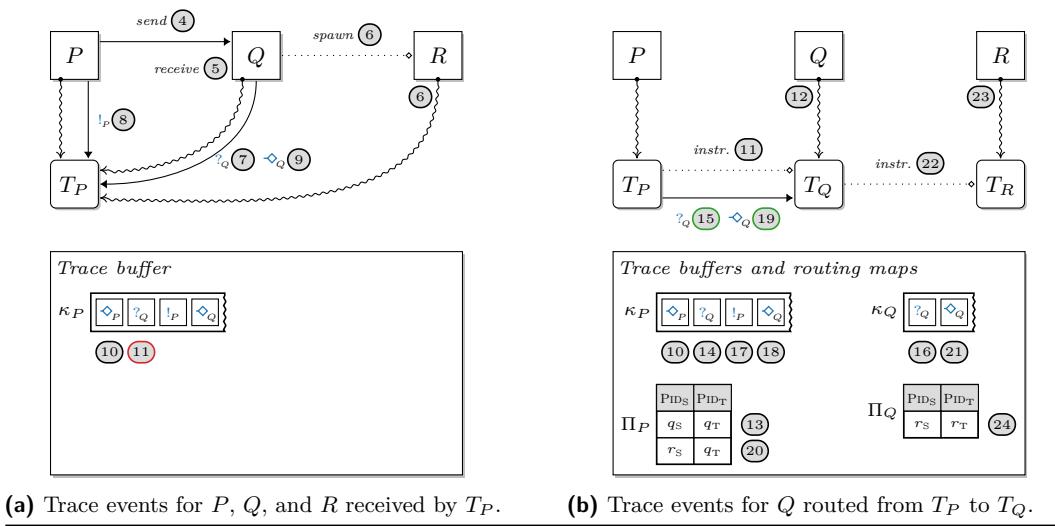


Figure 5 Next-hop trace event routing using tracer routing maps Π (monitors omitted).

process Q that is independent of the partition of P , step ⑤. Meanwhile, T_P receives the send event $\langle \text{evt}, !, p_s, q_s \rangle$ in step ⑩ after P messages Q in step ⑥ of fig. 4c. Subsequent \diamond events that T_P or T_Q may gather are handled as described in steps ③–⑤. Figs. 4c and 4d show how the final tracer T_R is instrumented in step ⑫ after Q spawns R in step ⑧. As before, T_Q traces R automatically in step ⑧. T_Q receives the event $\langle \text{evt}, \diamond, q_s, r_s, f_{s_R} \rangle$ generated by Q in step ⑪. T_R invokes $\text{PREEMPT}(r_s, r_T)$ to create the trace partition for R in step ⑬.

3.2 Ensuring complete traces

The asynchrony between the SuS and tracer processes can induce the interleaved execution shown in fig. 5, as an alternative execution to that shown in figs. 4b–4d. In fig. 5a, T_P is slow to handle \diamond_P it receives in ③ of fig. 4a and fails to instrument T_Q promptly. Consequently, the events $?_Q$ and \diamond_Q that Q exhibits are sent to T_P in steps ⑦ and ⑨ of fig. 5a. Step ⑪ shows the case where $\langle \text{evt}, ?, q_T \rangle$ is processed by T_P , rather than by the *intended* tracer T_Q that would have been instrumented by T_P . This error breaches the *completeness* property of trace soundness w.r.t. Q , as the events $?_Q$ and \diamond_Q meant for Q reach the wrong tracer T_P .

To address this issue, RIARC uses a next-hop routing approach, where tracers *retain* the events they should handle and *forward* the rest to neighbouring tracers. We use the term *dispatcher tracer* (*dispatcher* for short) to describe a tracer that receives trace events meant to be handled by another tracer. For instance, T_P in fig. 5a becomes the dispatcher tracer for Q when it receives the events $?_Q$ and \diamond_Q exhibited by Q , steps ⑦ and ⑨. We expect these events to be handled by T_Q once it is instrumented. Dispatchers are tasked with embedding trace event (evt) or detach requests (dtc) into routing packet messages (rtd) and transmitting them to the next *known* hop. In fig. 5b, T_P dispatches the events $?_Q$ and \diamond_Q as follows. It first instruments T_Q with Q in step ⑪. Next, T_P prepares $\langle \text{evt}, ?, r_s \rangle$ and $\langle \text{evt}, \diamond, q_s, r_s, f_{s_R} \rangle$ for transmission by embedding each in rtd messages (steps ⑭ and ⑯). T_P forwards the resulting routing packets, $\langle \text{rtd}, p_T, \langle \text{evt}, ?, r_s \rangle \rangle$ and $\langle \text{rtd}, p_T, \langle \text{evt}, \diamond, q_s, r_s, f_{s_R} \rangle \rangle$, to its next-hop neighbour T_Q in steps ⑮ and ⑯. The trace event $\langle \text{evt}, !, p_s, q_s \rangle$, however, is not forwarded but handled by T_P , as step ⑰ shows. Concurrently, T_Q acts on the forwarded events $?_Q$ and \diamond_Q in steps ⑯ and ⑰ and instruments T_R as a result, step ⑲.

Algorithm 1 Logic handling of trace events, detach request dispatching, and forwarding.

```

1  def LOOPo(σ, ξM)
2    forever do
3      m ← next message from trace buffer κ
4      match m.τ do
5        case evt: σ ← HANDLEVENTo(σ, ξM, m)
6        case dtc: σ ← DISPATCHDTC(σ, ξM, m)
7        case rtd: σ ← FORWRDRTDo(σ, ξM, m)
8
9  def HANDLEVTo(σ, ξM, e)
10   match e.ℓ do
11     case ◊: return HANDLSPWNo(σ, ξM, e)
12     case ∗: return HANDLEEXITo(σ, ξM, e)
13     case !, ?: return HANDLCOMMo(σ, ξM, e)
14
15  def HANDLSPWNo(σ, ξM, e)
16    match σ.Π(e.ιS) do
17      case ⊥: # No next-hop for e.ιS; handle e
18        ANALYSE.EVT(ξM, e)
19        σ ← INSTRUMENTo(σ, e, self())
20      case jT: # Next-hop for e.ιS exists via jT
21        DISPATCH(e, jT)
22        # Set next-hop of e.ιS to tracer of e.ιS
23        σ.Π ← σ.Π ∪ {⟨e.ιS, jTo(σ, ξM, e)
26    match σ.Π(e.ιS) do
27      case ⊥: # No next-hop for e.ιS; handle e
28        ANALYSE.EVT(ξM, e)
29        σ.Γ ← σ.Γ \ {⟨e.ιS, o⟩}
30        TRYGC(σ)
31      case jT: DISPATCH(e, jT)
32
33  def HANDLCOMMo(σ, ξM, e)
34    match σ.Π(e.ιS) do
35      case ⊥: ANALYSE.EVT(ξM, e)
36      case jT: DISPATCH(e, jT)
37
38  def DISPATCHDTC(σ, d)
39    match σ.Π(d.ιS) do
40      case ⊥: fail dtc next-hop must be defined
41      case jT:
42        DISPATCH(d, jT)
43        # Next-hop for d.ιS no longer needed
44        σ.Π ← σ.Π \ {⟨d.ιS, jTo(σ, r)
48    m ← r.m # Read embedded message in r
49    match m.τ do
50      case dtc: return FORWDDTC(σ, r)
51      case evt: return FORWDEVT(σ, r)
52
53  def FORWDDTC(σ, r)
54    d ← r.m
55    match σ.Π(d.ιS) do
56      case ⊥: fail dtc next-hop must be defined
57      case jT:
58        FORWD(r, jT)
59        # Next-hop for d.ιS no longer needed
60        σ.Π ← σ.Π \ {⟨d.ιS, jTS) do
66      case ⊥: fail evt next-hop must be defined
67      case jT:
68        FORWD(r, jT)
69        # For spawn events, tracer also sets a
70        # new next-hop for e.ιS
71        # Next-hop of e.ιS to same tracer of e.ιS
72        if (e.ℓ = ◊)
73          σ.Π ← σ.Π ∪ {⟨e.ιS, jT

```

Tracers determine the events to retain or forward using the routing map, $\Pi: \text{PID}_S \rightarrow \text{PID}_T$. Every tracer queries its private routing map for each message it receives on SuS PID $m.\iota_S$. A tracer forwards a message to its neighbouring tracer with PID ι_T if a next-hop for that message exists, i.e., $\Pi(m.\iota_S) = \iota_T$. When the next-hop is undefined, i.e., $\Pi(m.\iota_S) = \perp$, m is handled by the tracer. HANDLEVT, HANDLEEXIT and HANDLCOMM in alg. 1 implement this forwarding logic on lines 14, 23 and 31.

Dynamically populating the routing map is key to transmitting messages between tracers. A tracer adds the new mapping $e.\jmath_S \mapsto \jmath_T$ to its routing map Π in case 1 or 2 below whenever it processes spawn trace events $e = \langle \text{evt}, \diamond, \iota_S, \jmath_S, \xi_S \rangle$. One of two cases is considered for $e.\iota_S$:

Algorithm 2 Tracer instrumentation operations for direct (\circ) and priority (\bullet) modes.

Expect: $e = \langle \text{evt}, \diamond, \iota_s, j_s, \varsigma_s \rangle$ <pre> 1 def INSTRUMENT$_{\circ}$(σ, e, ι_T) 2 if (($\varsigma_M \leftarrow \sigma.\Lambda(e.\varsigma_S)$) $\neq \perp$) # New tracer j_T for new SuS process $e.j_s$ 3 $j_T \leftarrow \text{spwn}(\text{TRACER}(\sigma, \varsigma_M, e.j_s, \iota_T))$ 4 $\sigma.\Pi \leftarrow \sigma.\Pi \cup \{e.j_s, j_T\}$ 5 else # In \circ mode, this tracer has detached # all processes from its dispatcher ι_T # This tracer traces new SuS process $e.j_s$ # by tracing inheritance assumption A₂ 6 $\sigma.\Gamma \leftarrow \sigma.\Gamma \cup \{e.j_s, \circ\}$ 7 return σ</pre>	Expect: $e = \langle \text{evt}, \diamond, \iota_s, j_s, \varsigma_s \rangle$ <pre> 8 def INSTRUMENT$_{\bullet}$(σ, e, ι_T) 9 if (($\varsigma_M \leftarrow \sigma.\Lambda(e.\varsigma_S)$) $\neq \perp$) # New tracer j_T for new SuS process $e.j_s$ 10 $j_T \leftarrow \text{spwn}(\text{TRACER}(\sigma, \varsigma_M, e.j_s, \iota_T))$ 11 $\sigma.\Pi \leftarrow \sigma.\Pi \cup \{e.j_s, j_T\}$ 12 else # In \bullet mode, this tracer must detach # SuS process $e.j_s$ from its dispatcher ι_T 13 DETACH($e.j_s, \iota_T$) # This tracer traces new SuS process $e.j_s$ 14 $\sigma.\Gamma \leftarrow \sigma.\Gamma \cup \{e.j_s, \bullet\}$ 15 return σ</pre>
---	---

- $\Pi(\iota_s) = \perp$. The next-hop for e is undefined, which cues the tracer to instrument the SuS process with PID j_s . When applicable, the tracer processes the event *and* instruments a separate tracer with PID j_T . It then adds the mapping $e.j_s \mapsto j_T$ to Π . The tracer leaves Π *unmodified* and handles the event itself if a separate tracer is not required. Opting for a separate tracer is determined by the instrumentation map Λ , as discussed in sec. 3.5.
- $\Pi(\iota_s) = j_T$. The next-hop for e is defined, and the tracer forwards the event to the neighbouring tracer j_T . The tracer also records a new next-hop by adding $e.j_s \mapsto j_T$ to Π . The addition of $e.j_s \mapsto j_T$ in cases 1 and 2 ensures that future events originating from j_s can always be forwarded via a next-hop to a neighbouring tracer j_T (see invariants on lines 37, 51, and 60). Fig. 5b shows the routing maps of the tracers T_P and T_Q . T_P adds $q_s \mapsto q_T$ in step ⑯ after processing $\langle \text{evt}, \diamond, p_s, q_s, f_{s_Q} \rangle$ from its trace buffer in ⑮. T_P then instruments Q with the tracer T_Q in step ⑰; an instance of case 1. The function INSTRUMENT in alg. 2 details this on line 4, where the mapping $e.j_s \mapsto j_T$ is added to Π following the creation of tracer j_T , line 3. Step ⑳ of fig. 5b is an instance of case 2. Here, T_P adds $r_s \mapsto q_T$ to Π_P after processing $\langle \text{evt}, \diamond, q_s, r_s, f_{s_R} \rangle$ for R in step ⑱ since $\Pi_P(q_s) = q_T$. Crucially, T_P *does not* instrument a new tracer, but delegates the task to T_Q by forwarding \diamond_Q . Lines 20 and 64 in alg. 1 (and later line 24 in alg. 3) are manifestations of this, where the mapping $e.j_s \mapsto j_T$ is added after the \diamond event e is forwarded to the next-hop j_T . T_Q instruments the SuS process R in step ㉑ with T_R , which has the PID r_T . It then adds the mapping $r_s \mapsto r_T$ to Π_Q in step ㉒, as no next-hop is defined for q_s , i.e., $\Pi_Q(q_s) = \perp$. Henceforth, any events exhibited by R and received at T_P can be dispatched by the latter tracer through T_Q to T_R .

Note that every tracer is *only* aware of its neighbouring tracers. This means messages may pass through multiple tracers before reaching their intended destination. Next-hop routing keeps the logic inside RIARC straightforward since tracers forward messages based on local information in their routing map. This approach makes the instrumentation set-up adaptable to dynamic changes in the SuS and has been shown to induce lower latency when compared to general routing strategies [80, 104]. The DAG of interconnected tracers induced by next-hop routing ensures that every message is eventually delivered to the correct tracer if a path exists or handled by the tracer otherwise. Fig. 5b illustrates this concept, where the next-hop mappings inside Π_P point to T_Q , and the mappings in Π_Q point to T_R . Consequently, any events that R exhibits and that T_P receives are forwarded *twice* to reach the target tracer T_R : from tracer T_P to T_Q , and from T_Q to T_R . RIARC relies on the operations DISPATCH and FORWD to achieve next-hop routing (see [8, alg. 4 in app. A]). DISPATCH creates a routing packet, $\langle \text{rtd}, \iota_T, m \rangle$, and embeds the trace event or detach message m to be routed. Alg. 1 shows how tracers handle routing packets. For instance, FORWDEVT extracts the embedded

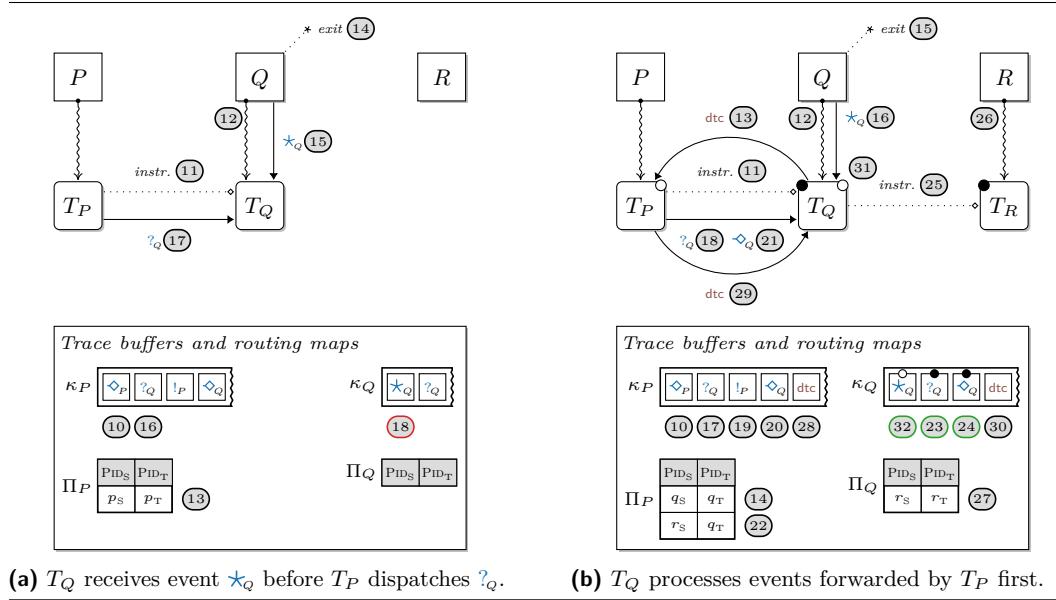
message from the routing packet on line 58 and queries the routing map to determine the next-hop, line 59. If found, the packet is forwarded, as $\text{FORWD}(r, j_T)$ on line 62 indicates. Crucially, the **fail** invariant on line 60 asserts that the next-hop for a routing packet is *always* defined. The cases for DISPATCHDTC and FORWDDTC in alg. 1 are analogous.

3.3 Ensuring consistent traces

Next-hop routing alone does not guarantee trace consistency, i.e., that the order of events in the trace reflects the one in which these occur locally at SuS processes, def. 1. Trace event reordering arises when a tracer gathers events of a SuS process (we call these *direct events*) and simultaneously receives *route events* concerning said process from other tracers. Fig. 6a gives another interleaving to the one of fig. 5b to underscore the deleterious effect such a race condition provokes when events are reordered at T_Q . In step ⑫ T_Q takes over T_P to continue tracing process Q . T_Q collects the event \star_Q in step ⑯, which happens before T_Q receives the routed event $?_Q$ concerning Q in step ⑰ of fig. 6a. If T_Q processes events from its trace buffer κ_Q in sequence, as in step ⑱, it violates trace consistency w.r.t. Q (the correct trace ordering should be “ $?_Q \cdot \star_Q \cdot \star_Q$ ”). Naïvely handling \star before $?$ erroneously reflects that Q receives messages after it terminates.

RIARC tracers resolve this issue by prioritising the processing of route events using selective message reception [42]. In doing so, tracers encode the invariant that “*route events* temporally precede all others that are gathered *directly* by the tracer”. RIARC tracers operate in one of two modes, priority (\bullet) and direct (\circ), which adequately distinguishes past (i.e., route) and current (i.e., direct) events from the perspective of the tracer receiving them.

Fig. 6b illustrates this concept. It shows that when in priority mode, T_Q dequeues the route events $?_Q$ and \star_Q labelled by \bullet first. The event $?_Q$ is handled in step ⑬, whereas \star_Q results in the instrumentation of tracer T_R in step ⑮ of fig. 6b. Meanwhile, T_Q can still receive events directly from Q while priority events are being handled. Yet, direct trace events from Q are considered only *after* T_Q transitions to direct mode. Newly-instrumented tracers default to \bullet mode to implement the described logic; see [8, line 14 in alg. 4 of app. A].



■ **Figure 6** Trace event reordering using priority (\bullet) and direct (\circ) tracer modes (monitors omitted).

Algorithm 3 Logic handling • trace events, detach request acknowledgements, and forwarding.

```

1  def LOOP•( $\sigma, \varsigma_M$ )
2  forever do
3     $r \leftarrow$  next rtd message from trace buffer  $\kappa$ 
4     $m \leftarrow r.m$  # Read embedded message in  $r$ 
5    match  $m.\tau$  do
6      case evt:  $\sigma \leftarrow \text{HANDLEVT}_•(\sigma, \varsigma_M, r)$ 
7      case dtc:
8        # dtc ack relayed from dispatch tracer
9         $\sigma \leftarrow \text{HANDLDTC}(\sigma, \varsigma_M, r)$ 

10   def HANDLEVT•( $\sigma, \varsigma_M, r$ )
11    $e \leftarrow r.m$ 
12   match  $e.\ell$  do
13     case  $\diamond$ : return HANDLSPWN•( $\sigma, \varsigma_M, r$ )
14     case  $\star$ : return HANDLEXIT•( $\sigma, \varsigma_M, r$ )
15     case  $!?$ : return HANDLCOMM•( $\sigma, \varsigma_M, r$ )
16
17   def HANDLSPWN•( $\sigma, \varsigma_M, r$ )
18    $e \leftarrow r.m$ 
19   match  $\sigma.\Pi(e.\iota_S)$  do
20     case  $\perp$ : # No next-hop for  $e.\iota_S$ ; handle  $e$ 
21       ANALYSE.EVT( $\varsigma_M, e$ )
22        $\iota_T \leftarrow r.\iota_T$  # Read PID of dispatch tracer
23        $\sigma \leftarrow \text{INSTRUMENT}_•(\sigma, e, \iota_T)$ 
24     case  $\jmath_T$ : # Next-hop for  $e.\iota_S$  exists via  $\jmath_T$ 
25       FORWD( $r, \jmath_T$ )
26       # Set next-hop of  $e.\jmath_S$  to tracer of  $e.\iota_S$ 
27        $\sigma.\Pi \leftarrow \sigma.\Pi \cup \{\langle e.\jmath_S, \jmath_T \rangle\}$ 
28
29   return  $\sigma$ 

30   def HANDLEXIT•( $\sigma, \varsigma_M, r$ )
31    $e \leftarrow r.m$ 
32   match  $\sigma.\Pi(e.\iota_S)$  do
33     case  $\perp$ : # No next-hop for  $e.\iota_S$ ; handle  $e$ 
34       ANALYSE.EVT( $\varsigma_M, e$ )
35        $\sigma.\Gamma \leftarrow \sigma.\Gamma \setminus \{\langle e.\iota_S, \bullet \rangle\}$ 
36       TRYGC( $\sigma$ )
37     case  $\jmath_T$ : FORWD( $r, \jmath_T$ )
38
39   return  $\sigma$ 

40   def HANDLCOMM•( $\sigma, \varsigma_M, r$ )
41    $e \leftarrow r.m$ 
42   match  $\sigma.\Pi(e.\iota_S)$  do
43     case  $\perp$ :
44       assert  $d.\iota_T = \text{self}()$  unexpected dtc ack
45        $\sigma.\Gamma \leftarrow (\sigma.\Gamma \setminus \{\langle d.\jmath_S, \bullet \rangle\}) \cup \{\langle d.\jmath_S, \gamma \rangle\}$ 
46       if  $(\{\langle \iota_S, \gamma \rangle \mid \langle \iota_S, \gamma \rangle \in \sigma.\Gamma, \gamma = \bullet\} = \emptyset)$ 
47         LOOP○( $\sigma, \varsigma_M$ ) # Put tracer in ○ mode
48
49     case  $\jmath_T$ :
50       assert  $d.\iota_T \neq \text{self}()$  dtc meant for  $\iota_T$ 
51       FORWD( $r, \jmath_T$ )
52
53   return  $\sigma$ 
```

LOOP_• in alg. 3 shows the logic prioritising routed events, which are dequeued on line 3 and handled on line 6. HANDLSPWN, HANDLEXIT, and HANDLCOMM in LOOP_○ and LOOP_• handle events *differently*. A tracer in direct mode performs *one* of three actions (see alg. 1):

1. it *analyses* events for RV purposes via the function ANALYSE.EVT(ς_M, e), e.g. line 32,
2. it *dispatches* events that it directly gathers using DISPATCH(e, \jmath_T), when events ought to be handled by other tracers, e.g. line 33, or
3. it *forwards* routed events to the next-hop through FORWD(r, \jmath_T), e.g. line 62.

Tracers in priority mode exclusively handle routed messages as points 1 and 3 describe, e.g. lines 38 and 39 in alg. 3. However, no event dispatching is performed.

3.4 Isolating tracers

A tracer in priority mode coordinates with the dispatch tracer of a particular SuS process it traces. This enables the tracer to determine when *all* of the events of that process have been routed to it by the dispatch tracer. The negotiation is effected using dtc, which the tracer sends to the relevant dispatch tracer. Each tracer records the set of processes it traces in the *traced-processes map*, $\Gamma : \text{PID}_S \rightarrow \{\circ, \bullet\}$. A SuS process mapping is added to Γ when a tracer starts gathering trace events for that process and removed once the process terminates. Lines 6 and 14 in alg. 2 add fresh mappings to Γ ; lines 26 in alg. 1 and 31 in alg. 3 purge mappings from Γ . A tracer in priority mode must issue a dtc request *for each* process it

tracks in Γ before it can transition to direct mode and start operating on the trace events it gathers directly. The detach request, $d = \langle \text{dtc}, \iota_T, \iota_S \rangle$, contains the PIDs of the issuing tracer and the SuS process to be detached from the dispatch tracer. Once the tracer receives an acknowledgement to the **dtc** request for the SuS PID $d.\iota_S$ from the dispatch tracer, it updates the corresponding entry $d.\iota_S \mapsto \bullet$ in Γ , marking it as detached, $d.\iota_S \mapsto \circ$. Alg. 3 shows this logic on line 46. A tracer transitions from priority to direct mode once *all* the processes in its Γ map are marked detached; line 47 in alg. 3 performs this check. Once in direct mode, tracers are isolated from others in the choreography.

Fig. 6b depicts the tracer T_Q in priority mode sending the detach request $\langle \text{dtc}, q_T, q_S \rangle$ for SuS PID q_S to the dispatch tracer. This happens in step ⑯, after T_Q starts tracing Q directly in step ⑮. Alg. 2 effects this transaction with the dispatch tracer by the operation DETACH on line 13; see [8, app. A] for definition of DETACH. The **dtc** request issued by T_Q is deposited in the trace buffer of the dispatch tracer T_P after the events ?_Q and ?_Q . T_P processes the messages in its buffer sequentially in ⑩, ⑯, ⑰, ⑱ and ⑲, and forwards ?_Q and ?_Q to T_Q , steps ⑯ and ⑰. Crucially, T_P acknowledges the **dtc** request issued by T_Q : T_P dispatches **dtc** back to tracer T_Q , as step ⑳ indicates. T_Q first handles the events ?_Q and ?_Q (tagged with \bullet in fig. 6b) in steps ㉑ and ㉒. Lastly, T_Q handles **dtc** in ㉓ and marks process Q as detached from its dispatch tracer T_P . The update on the traced-process map Γ is performed by HANDLDTC on line 46 in alg. 3. Tracer T_Q in fig. 6b transitions to direct mode in step ㉔, when the only process Q that it traces is detached. T_Q resumes handling ?_Q in step ㉕, which is consistent w.r.t. the events exhibited locally at Q , i.e., “ $\text{?}_Q . \text{?}_Q . \star_Q$ ”.

An acknowledgement to a detach request sent from a dispatch tracer, $\langle \text{dtc}, \iota_T, \iota_S \rangle$, is generally propagated through multiple next-hops before it reaches the tracer with PID ι_T issuing the request. Since a **dtc** request informs the dispatch tracer that ι_T is gathering trace events for the SuS PID ι_S *directly*, the next-hop entries in the routing maps of tracers on the DAG path from the dispatch tracer to ι_T are *stale*. Each tracer on this DAG path purges the next-hop entry for the SuS PID ι_S in Γ once it forwards **dtc** to the neighbouring tracer. DISPATCHDTC and FORWDDTC in alg. 1 perform this clean-up. Fig. 6b does not illustrate the latter clean-up flow, which we summarise next. After receiving **dtc**, the dispatch tracer T_P removes from Π_P the next-hop mapping $q_S \mapsto q_T$ and calls DISPATCHDTC to acknowledge the detach request $\langle \text{dtc}, q_T, q_S \rangle$ it receives from T_Q . Similarly, T_P removes $r_S \mapsto q_T$ once it acknowledges the detach request $\langle \text{dtc}, r_T, r_S \rangle$ sent from T_R . Once T_Q receives the routing packet $\langle \text{rtd}, p_T, \langle \text{dtc}, r_T, r_S \rangle \rangle$ that embeds the detach acknowledgement T_P sends, it removes the next-hop mapping $r_S \mapsto r_T$ from Π_Q . T_Q then forwards this **dtc** acknowledgement to T_R .

RIARC ensures that all routing packets carrying **dtc** acknowledgements terminate at the tracers that issued these **dtc** requests. This requires *one* of two tracer conditions to hold:

1. either the tracer cannot forward the **dtc** acknowledgement to a next-hop, meaning that the tracer sent the **dtc** request, or
2. the tracer can forward the **dtc** acknowledgement via a next-hop, in which case the tracer did not issue the **dtc** request.

Alg. 3 enforces this invariant on lines 44 and 45 for case 1, and on lines 49 and 50 for case 2.

3.5 Minimising overhead

Instrumenting specific processes – in contrast to fully instrumenting the SuS – reduces the volume of gathered trace events and helps lower the runtime overhead induced. RIARC uses the instrumentation map, $\Lambda : \text{SIG}_S \rightarrow \text{SIG}_M$, to this end. Λ specifies the SuS function signatures to instrument and the corresponding RV monitor signatures tasked with the analysis via ANALYSEEVT. RIARC utilises the signature $e.\varsigma_S$ carried by spawn events $e = \langle \text{evt}, \text{?}, \iota_S, \jmath_S, \varsigma_S \rangle$ to

determine whether the SuS process spawning $e.\varsigma_s$ requires a separate tracer. The INSTRUMENT operations in alg. 2 perform this check against Λ (lines 2 and 9). If a separate tracer is not required, $e.\jmath_s$ is instrumented using the tracer of its parent process, $e.\iota_s$; see tracing assumptions A₁ and A₂. This logic caters for all the set-ups shown in figs. 1b, 1c, and 2b.

3.6 Shrinking the set-up

RIARC remains elastic by discarding unneeded tracers. Tracers in direct and priority mode purge SuS PID references from the traced-process map when handling \star trace events. HANDLEEXIT_o and HANDLEEXIT_• implement this logic in algs. 1 and 3 on lines 26 and 31. Tracer termination does *not* occur when the tracer has no processes left to trace, i.e., when $\Gamma = \emptyset$, since the tracer may be required to forward trace events to neighbouring tracers. Instead, tracers perform a garbage collection check each time a mapping from Γ or Π is removed. A tracer terminates when $\Gamma = \Pi = \emptyset$, indicating that it has no SuS processes left to trace or any next-hop forwarding to perform. TRYGC used on lines 27, 41, and 55 in alg. 1, as well as on line 32 in alg. 3 encapsulates this check. Note that garbage collection never prematurely disrupts the RV analysis that tracers conduct, as invocations to ANALYSE.EVT always precede TRYGC checks in our logic of algs. 1 and 3.

4 Correctness validation

We assess the validity of RIARC in two stages. First, we confirm its implementability by instantiating the core logic of algs. 1–3 to Erlang. Our implementation targets two RV scenarios: online and offline monitoring [64, 22]. Second, we subject the implementation to a series of systematic tests using a selection of instrumentation set-ups. These tests exhaustively emulate the interleaved execution of the SuS and tracer processes by generating all the *valid* permutations of events in a set of traces. This exercises the tracer choreography invariants mentioned in sec. 3, confirming the integrity of the tracer DAG topology under each interleaving. We also use specialised RV monitor signatures in ANALYSE.EVT to assert the soundness (def. 1) of trace event sequences analysed by tracers; see algs. 1 and 3 in sec. 3.

4.1 Implementability

Our implementation of RIARC maps the tracer processes from sec. 3 to Erlang actors. The routing (Π), instrumentation (Λ), and traced-processes (Γ) maps constituting the tracer state σ are realised as Erlang maps for efficient access. Trace event buffers κ coincide with actor mailboxes, while the remaining logic in algs. 1–3 translates directly to Erlang code. This one-to-one mapping gives us confidence that our implementation reflects the algorithm logic.

In *online* RV, monitors analyse trace events while the SuS executes, whereas the *offline* setting defers this analysis until the system terminates; [8, fig. 11 in app. B.1] captures the distinction in process tracing between online and offline instrumentation in our setting (showing trace buffers only). The online instrumentation set-up employs the tracing infrastructure offered by the EVM, which deposits SuS trace event messages in tracer mailboxes. Erlang tracing complies with tracing assumption A₁, enabling RIARC to instrument disjoint SuS processes sets. We configure the EVM with the `set_on_spawn` flag so that spawned processes automatically inherit the same tracer as their parent [42]. This tracer assignment is atomic, meeting tracing assumption A₂. We also use the `procs`, `send`, and `receive` tracing flags, which constrain the events emitted by the EVM to \diamond , \star , $!$, and $\#$. The EVM enforces single-process tracing, i.e., tracing assumption A₃, and guarantees that \diamond events of descendant processes are causally-ordered [128], i.e., tracing assumption A₄.

The offline counterpart differs only in its tracing layer, where events are read as *recorded* runs of the SuS. Recorded runs can be acquired externally, e.g. using DTrace [36] or LTTng [56], making it possible to monitor systems that execute outside of the EVM. Our bespoke offline tracing engine of [8, fig. 11b in app. B.1] fulfils tracing assumptions A₁–A₄. This is crucial since it permits the *same* implementation of RIARC to be used in online and offline settings. Sec. 4.2 leverages this aspect to validate RIARC exhaustively using trace permutations.

We develop two versions of the TRACE, CLEAR, and PREEMPT functions of [8, alg. 5 in app. A] to standardise tracing for online and offline use. The overloads for online use access the EVM tracing via the Erlang built-in primitive `trace` [42]. The second set of overloads wraps around our offline tracing engine to replay files containing specifically-formatted trace events. Offline tracing relaxes tracing assumption A₄, as recorded runs do not generally guarantee that the \diamond events of descendant SuS processes are causally ordered. Our offline tracing logic relies on the PID information carried by \diamond events to rearrange them and recover the causal ordering per tracing assumption A₄. $\text{TRACE}(\iota_S, \iota_T)$ registers a tracer ι_T with the offline tracing engine, which maintains an event buffer for ι_T , together with a set of SuS PIDs that ι_T traces. A tracer can use TRACE with multiple SuS PIDs to register to obtain events for a process set, i.e., tracing assumption A₁. The tracing engine accumulates the events it reads from file in each tracer buffer and delivers events to the corresponding tracer mailbox once the causal ordering between \diamond events of descendant SuS processes is established. Our offline tracing engine implements tracing inheritance (tracing assumption A₂) and enforces single-process tracing (tracing assumption A₃); [8, ex. 7 in app. B.1] sketches how the tracing engine uses its internal tracer buffers to deliver events to tracers.

4.2 Correctness

Conventional testing does not guarantee the absence of concurrency errors due to the different interleaved executions that may be possible [105]. While subjecting the system under test to high loads raises the likelihood of obtaining more coverage, this still depends on external factors, such as scheduling, which dictate the executions induced in practice. Controlling the conditions for concurrency testing requires a *systematic exploration* of all the interleaved executions [74]. In fact, it is *not the size* of the testing load that matters, but the choice of interleaved executions that exhaust the space of possible system states [14]. Concuerror [48] is a tool for systematic Erlang code testing. Unfortunately, we could not use Concuerror to test our RIARC implementation, as we were unable to integrate it with Erlang tracing.

We, nevertheless, adopt the systematic scheme advocated by Concuerror. Our approach uses the offline tracing tool described in sec. 4.1 to induce specific interleaved sequences for instrumentation set-ups, such as those of figs. 1b, 1c, and 2a. We obtain these sequences by taking all the sound (def. 1) event permutations of traces produced by the SuS. These sequences are then replayed by the offline tracing engine to systematically induce interleaved SuS executions. Our final RIARC implementation embeds further invariants besides those mentioned in sec. 3, e.g. the `assert` and `fail` statements in algs. 1 and 3. Readers are referred to [8, app. B.2] for the full list. We ascertain *trace soundness* for each SuS interleaving that is emulated. This is accomplished via the function ANALYSE_EVT, which we preload with monitors that assert the event sequence expected at each tracer. We also use identical tests in our empirical evaluation of sec. 5 under high loads. It is worth mentioning that while we systematically drive the execution of the SuS, we do not control the execution of tracers. Yet, we indirectly induce various dynamic tracer arrangements in the monitor DAG topology under the different groupings of SuS process sets that tracers instrument. For example, we fully instrument system depicted in fig. 2a in all its configurations, e.g. $\mathcal{C}_1 = [T_{\{P\}} \rightsquigarrow$

$\{P\}, T_{\{Q\}} \rightsquigarrow \{Q\}, T_{\{R\}} \rightsquigarrow \{R\}$, $C_2 = [T_{\{P,Q\}} \rightsquigarrow \{P,Q\}, T_{\{R\}} \rightsquigarrow \{R\}]$, \dots , $C_5 = [T_{\{P,Q,R\}} \rightsquigarrow \{P,Q,R\}]$, as well as instrument it partially, e.g. $C_6 = [T_{\{P\}} \rightsquigarrow \{P\}]$, $C_7 = [T_{\{P,Q\}} \rightsquigarrow \{P,Q\}]$, etc. Each of these configurations, when individually paired with every fabricated interleaved execution of the SuS, indicate that our RIARC implementation and corresponding logic of sec. 3 is correct.

5 Empirical evaluation

We assess the feasibility of our RIARC implementation, confirming it safeguards the *responsive*, *resilient*, *message-driven*, and *elastic* attributes of the SuS. Sec. 4 targets a small selection of instrumentation set-ups to induce interleaved execution sequences and validate correctness exhaustively. We now employ *stress testing* [109] to investigate how RIARC performs in terms of the *runtime overhead* it exhibits. Our study focusses on *online* monitoring, as its overhead requirement is far more stringent than offline monitoring [63, 64, 22, 71]. We evaluate RIARC against inline instrumentation since the latter is regarded as the most efficient instrumentation technique [62, 61, 22]. This comparison establishes a solid basis for our results to be generalised reliably. We also compare RIARC to centralised instrumentation to confirm that the latter approach does not scale under typical loads.

Our experiments are extensive. We use two hardware platforms to model edge-case scenarios based on limited hardware and general-case scenarios using commodity hardware. The evaluation subjects inline, centralised, and RIARC instrumentation to high loads that go beyond the state of the art and use realistic workload profiles. We gauge overhead under three performance metrics, the *response time*, *memory consumption*, and *scheduler utilisation*, which are crucial for reactive systems [7, 109]. Our results confirm that the overhead RIARC induces is adequate for applications such as soft real-time systems [42, 94], where the latency requirement is typically in the order of seconds [92]. We also show that RIARC yields overhead comparable to inlining in settings exhibiting moderate concurrency.

5.1 Benchmarking tool

Benchmarking is standard practice for gauging runtime overhead in software [100, 77, 35]. Frameworks, including DaCapo [28] and Savina [84], offer limited concurrency, making them inapplicable to our case; see [8, app. C.1] for detailed reasons. Industry-proven *synthetic* load testing benchmarking tools cater to reactive systems, e.g. Apache JMeter [67], Tsung [115], and Basho Bench [23]. Their general-purpose design, however, necessarily treats systems as a black box by gathering metrics externally, which may impact measurement *precision* [7]. Moreover, these load testers generate standard workloads, e.g. Poisson processes [79, 102, 89], but lack others, e.g. load bursts, that replicate typical operation or induce edge-case stress.

We adopt BenchCRV [7], another synthetic load testing tool specific to RV benchmarking for reactive systems. BenchCRV sets itself apart from the tools mentioned above because it does not require external software (e.g., a web server) to drive tests. Instead, BenchCRV produces different SuS models that *closely emulate* real-world software behaviour. These models are based on the master-worker paradigm [120]: a pervasive architecture in distributed (e.g. Big Data stream processing frameworks, render farms) and concurrent systems [129, 73, 55, 132]. Like Tsung and Basho Bench, BenchCRV exploits the lightweight EVM process model to generate highly-concurrent synthetic workloads.

BenchCRV creates master-worker models and induces workloads derived from configurable parameters. In these models, the master process spawns a series of workers and allocates tasks. The volume of workers per benchmark run is set via the parameter n . Each worker task consists of a *batch* of requests that the worker receives, processes, and echoes back to

the master process. The amount of requests batched in one task is given by the parameter w . Workers terminate when all of their allotted tasks are processed and acknowledged by the master. BenchCRV creates workers based on *workload profiles*. A profile dictates how the master spreads its creation of workers along the loading timeline, t , given in seconds. BenchCRV supports three workload profiles based on ones typical in practice:

Steady models the SuS under stable workload (Poisson process).

Pulse models the SuS under gradually rising and falling workload (Normal distribution).

Burst models the SuS under stress due to workload spikes (Log-normal distribution).

BenchCRV records three performance metrics to give a multi-faceted view of system overhead:

Mean response time in milliseconds (ms), gauging monitoring latency effects on the SuS.

Mean memory consumption in GB, gauging monitoring memory pressure on the SuS.

Mean scheduler utilisation as a percentage of the total processing capacity, showing how monitors maximise the scheduler use.

The prevalent use of the master-worker paradigm, the veracity with which BenchCRV models systems, the range of realistic workload profiles, and the choice of runtime metrics it gathers make this tool ideal for our experiments. We refer readers to [8, app. C.2] and [7] for details.

5.2 Benchmark configuration

The BenchCRV master-worker models we generate take the role of the SuS in our experiments. We consider *edge-case* and *general-case* hardware platform set-ups for the following reasons:

P_E **Edge-case** captures platforms with *limited* hardware. It uses an Intel Core i7 M620 64-bit CPU with 8GB of memory, running Ubuntu 18.04 LTS and Erlang/OTP 22.2.1.

P_G **General-case** captures platforms with *commodity* hardware. It uses an Intel Core i9 9880H 64-bit CPU with 16GB of memory, running macOS 12.3.1 and Erlang/OTP 25.0.3.

The EVMs on platforms P_E and P_G are set with 4 and 16 scheduling threads, respectively. These scheduler settings coincide with the processors available on each SMP [12] platform. We also use the P_E and P_G platforms with two concurrency scenarios for reactive systems:

C_H **High concurrency scenarios** perform short-lived tasks, e.g. web apps that fulfil thousands of HTTP client requests by fetching static content or executing back-end commands.

C_M **Moderate concurrency scenarios** engage in long-running, computationally-intensive tasks, e.g. Big Data stream processing frameworks.

Our benchmark workloads match the hardware capacity afforded by P_E and P_G:

High concurrency benchmarks on P_E set $n = 100k$ workers and $w = 100$ work requests per worker. These generate $\approx (n \times w \text{ requests} \times w \text{ responses}) = 20M$ message exchanges between the master and worker processes, totalling $\approx (20M \times ! \text{ events} \times ? \text{ events}) = 40M$ analysable trace events. Platform P_G sets $n = 500k$ workers batched with $w = 100$ requests to produce $\approx 100M$ messages and $\approx 200M$ trace events. The high concurrency model C_H is studied in sec. 5.4.

Moderate concurrency benchmarks on P_G set $n = 5k$ workers and $w = 10k$ work requests per worker. These settings yield roughly the same number of trace events as on P_G with concurrency scenario C_H. The moderate concurrency model C_M is studied in sec. 5.5.

All experiments in secs. 5.4 and 5.5 use a total loading time of $t = 100s$. Each experiment consists of *ten* benchmarks that apply Steady, Pulse, and Burst workloads. We repeat every experiment *thrice* to obtain *negligible variability* and ensure the accuracy of our results; see [8, app. C.4] for a summary of these workloads and [8, app. C.5] for the precautions we take.

The hardware, OS, and Erlang versions of platforms P_E and P_G, combined with the workloads of concurrency scenarios C_H and C_M provide generality to our conclusions.

5.3 Instrumentation configuration

One challenge in conducting our experiments is the lack of RV monitoring tools targeting the EVM. To the best of our knowledge [64, tables 3 and 4], detectEr [72, 18, 19, 17, 70, 40] is the only RV tool for Erlang that implements centralised outline instrumentation². We are unaware of inline RV tools besides [38] and [3, 4]. Since the former tool is *unavailable*, we use the latter, more recent work³. In our experiments, we instrument the master *and each* worker process in the SuS models generated from sec. 5.2 to exert the highest possible load and capture *worst-case* scenarios. BenchCRV annotates work requests and responses with a unique sequence number to account for each message in benchmark runs. We leverage this numbering to write specialised monitor replicas that ascertain the *soundness* of trace event sequences reported to every RV monitor linked with the master and workers; see [8, app. C.5] for details. Equally crucial, this runtime checking introduces a degree of *realistic* RV analysis slowdown that is *uniform* across all monitors in the inline, centralised, and RIARC monitoring set-ups. We empirically estimate this slowdown at $\approx 5\mu\text{s}$ per analysed event.

5.4 High concurrency benchmarks

We study runtime overhead in the high concurrency scenario C_H with two aims. First, we show the effect overhead has on the SuS as it executes. Specifically, we consider how the memory consumption and scheduler utilisation impact the *latency* a client of the SuS experiences, e.g. end-user or application. We use the edge-case platform P_E for these experiments; analogous results obtained on P_G are detailed in [8, app. C]. Our second goal targets the general-case platform P_G to assess the *scalability* of the instrumentation methods through their optimal use of the *additional* memory and scheduler capacity afforded by P_G .

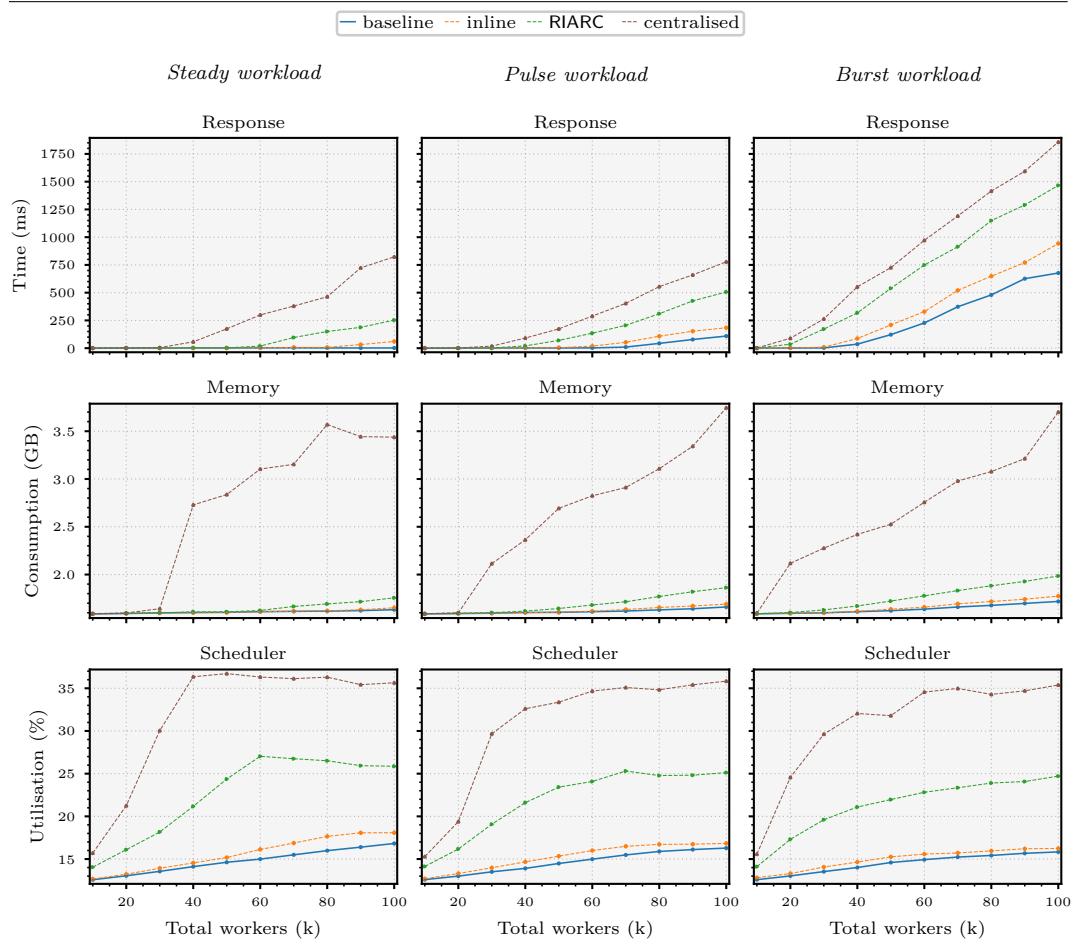
The charts in secs. 5.4.1–5.4.3 plot performance metrics, e.g. memory consumption (y -axis) against the number of concurrent worker processes or the execution duration (x -axis). Since inline instrumentation prevents us from delineating the SuS and monitoring-induced runtime overhead, we follow the standard RV literature practice and include the *baseline* plots, e.g. [19, 72, 46, 38, 99, 114, 112]. Baseline plots show the *unmonitored* SuS to compare the relative overhead between each evaluated instrumentation method.

5.4.1 Instrumentation overhead

The first set of experiments isolates the instrumentation overhead induced on the SuS: this is the aggregated cost of tracing *and* reporting the traces soundly per def. 1 to RV monitors. Crucially, these experiments *omit monitors*, as we want to quantify the instrumentation overhead and understand its impact on the SuS. This enables us to focus on the differences between inlining – regarded as the most efficient instrumentation method [62, 61, 22] – and outlining. As far as we know [64, 71], outlining has *never* been used for decentralised RV in a *dynamic* setting such as ours. While we confirm that inline instrumentation uses less memory and scheduler capacity, RIARC dynamically scales and economises their use *without* adverse impact on the latency. In fact, the latency induced by RIARC is a mere 519ms higher than that of inline instrumentation at the peak stress-inducing loading point of 3.7k workers/s under Burst workloads. Our experiments indicate that centralised instrumentation manages resources poorly due to its inability to scale, increasing the chances of failure; see sec. 5.4.2.

² <https://bitbucket.org/duncanatt/detecter-lite>

³ <https://github.com/ScienceofComputerProgramming/SCICO-D-22-00294>



■ **Figure 7** Isolated instrumentation overhead (*high workload, 100k workers*).

Fig. 7 plots our results. Centralised instrumentation carries the largest overhead penalty. Regardless of the workload applied, it uses the most memory, $\approx 3.8\text{GB}$, highlighting its ineptitude to scale. This stems from the backlog of trace event messages that accumulate in the mailbox of the central tracer and is a manifestation of two aspects. First, the central tracer does not consume events at the same rate worker processes produce them. Evidence of this *bottleneck* is visible as high scheduler utilisation in fig. 7 (bottom). This values settles at $\approx 36\%$ for the benchmarks with $\approx 40\text{k}$ workers under the Steady workload and $\approx 60\text{k}$ workers under Pulse and Burst workloads. Interpreting these $< 36\%$ scheduler usage values in isolation may suggest that centralised instrumentation has the potential to scale. However, its memory consumption plots in fig. 7 (middle) contradict this erroneous hypothesis.

By contrast, RIARC uses fewer resources to yield lower response times across the three workloads. The scheduler utilisation for RIARC slightly plateaus in the Steady ($\approx 60\text{k}$ workers) and Pulse ($\approx 70\text{k}$ workers) workload charts. This is not owed to scalability limitations of RIARC but to the intrinsic throttling instigated by the master process [120]. In fact, the plots for the baseline system and inline instrumentation in fig. 7 (middle) exhibit analogous signs of throttling. Even at a peak Burst workload of 3.7k workers/s, inline and RIARC instrumentation consume fairly similar amounts of memory, 1.7GB vs. 1.9GB , respectively.

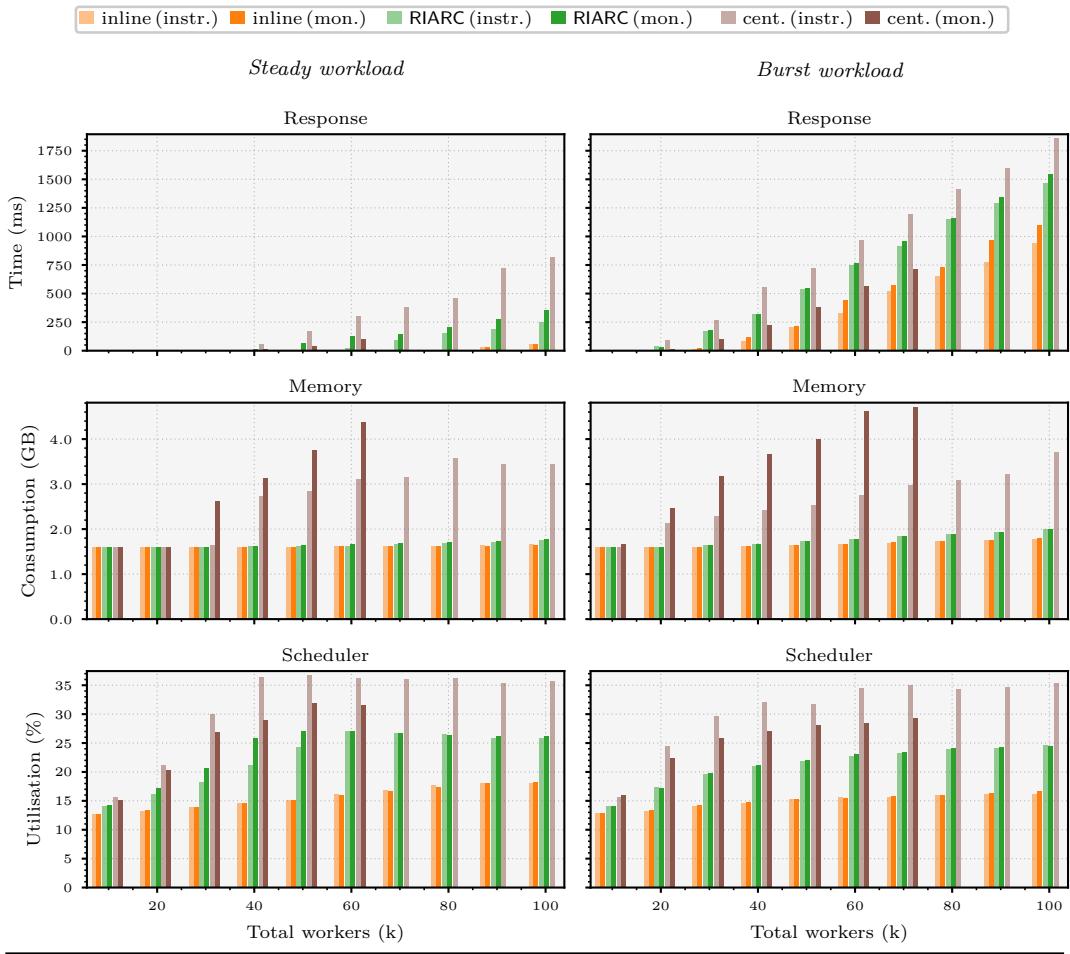


Figure 8 Instrumentation and RV monitoring overhead gap (*high* workload, 100k workers).

5.4.2 Monitoring overhead

Our second set of experiments extends the results of sec. 5.4.1 and quantifies the cost of RV monitoring. The *runtime monitoring* overhead combines the instrumentation and slowdown due to the RV analysis, established at $\approx 5\mu\text{s}$ per event in sec. 5.3 for our experiments. Fig. 8 plots the instrumentation (*instr.*) overhead from sec. 5.4.1 next to the runtime monitoring overhead (*mon.*). It shows that the RV analysis slowdown aggravates centralised monitoring to the point of crashing. Inline and RIARC monitoring are minimally affected. Our results also reveal that the instrumentation incurs the *major* overhead portion, not the RV analysis. Sec. 5.6 comments on this finding in the context of existing RV tools.

Fig. 8 plots our results under the Steady and Burst workloads; [8, fig. 14 in app. C.6.1] includes all three workloads. The charts for centralised monitoring exhibit a significant disparity between the instrumentation and runtime monitoring bar plots as the workload increases. This trend is consistent across both workloads in fig. 8. The lack of scalability of centralised monitoring in fig. 8 manifests as an increase in memory consumption but stabilised scheduler usage, as in fig. 7. Memory consumption and scheduler usage for centralised monitoring grow rapidly beyond $\approx 30\text{k}$ and $\approx 20\text{k}$ workers under the Steady and Burst workloads, respectively. Bottlenecks led our experiments to crash (shown as missing

bar plots in fig. 8). Crashes occur at $\approx 70k$ workers under the Steady and at $\approx 80k$ under Burst workload. By analysing the resulting dumps, we could attribute these crashes to memory exhaustion, which caused the EVM to fail. The dumps indicate severe memory pressure due to the vast backlog of trace event messages in the mailbox of the central tracer.

Inline and RIARC monitoring scale to accommodate the RV analysis slowdown. This is confirmed by cross-referencing the memory consumption and scheduler utilisation in fig. 8 for both monitoring methods. Each displays comparable overhead in their respective instrumentation and corresponding runtime monitoring bar plots. Fig. 8 (top) shows that inline and RIARC monitoring increase the latency, albeit for different reasons. The internal operation of RIARC enables us to deduce that its latency stems from message routing and dynamic tracer reconfiguration. Its scheduler utilisation plots support this observation. The latency due to inlining is a direct effect of RV analysis slowdown, provoked by the lock-step execution of monitors and the SuS. Other works, e.g. [46, 37], offer similar observations.

Dissecting our results uncovers further subtleties. The optimal scheduler utilisation of RIARC implies that its monitors are only active when triggered by trace events but remain idle otherwise. This inference is supported by the absence of sudden or continued memory growth for RIARC in fig. 8 (middle). The instrumentation and runtime monitoring latency bar plots for inline monitoring exhibit a growing pairwise gap that starts at $\approx 80k$ workers in fig. 8 (top right). The respective gap for RIARC at this mark is perceptibly lower. We credit this lower latency gap to outlining, which absorbs the slowdown effect of RV analyses. This leads us to conjecture that RIARC could accommodate monitors that perform richer RV analyses with minimal impact on the SuS. Our calculations from fig. 8 (top right) put the latency at 1093ms for inline monitoring vs. 1547ms for RIARC at a peak Burst workload of 3.7k workers/s: a 454ms difference, which is *lower* than the 519ms gap measured in sec. 5.4.1. Sec. 5.5 shows this gap is negligible in moderate concurrency scenarios.

5.4.3 Resource usage

We employ platform P_G with high concurrency C_H to confirm that our observations about inline and RIARC monitoring transfer to general cases. Secs. 5.4.1 and 5.4.2 deem centralised monitoring to be impractical. We, thus, omit it from the sequel; see [8, app. C.6.3] for results.

Our experiments now use 16 scheduling threads, $n = 500k$ workers, and $w = 100$ requests per worker, producing $\approx 100M$ messages and $\approx 200M$ trace events; [8, fig. 13 in app. C.4] render these Steady, Pulse, and Burst workload models. Secs. 5.4.1 and 5.4.2 bound the memory and scheduler metrics to the period the SuS executes to portray the *actual overhead* impact on the system. We refocus that view to assess the monitoring overhead in *its entirety* – from the point of SuS launch until monitors complete their RV analysis. Doing so reveals how inline and RIARC monitoring optimise the use of added memory and processing capacity. Results show that inline and RIARC monitoring are elastic and dynamically adapt to changes in the applied workloads; [8, app. C.6.3] confirms that centralised monitoring lacks this trait.

Fig. 9 gives a complete benchmark run under the Steady and Burst workloads. We relabel the x -axis with the benchmark duration and omit the response time plots since response time is inapplicable to these experiments (latency is an attribute of the SuS, not the monitors). In this run, the Steady workload generates a sustained load of $\approx 5k$ workers/s whereas Burst peaks at $\approx 17.8k$ workers/s under maximum load at $\approx 5s$; see [8, fig. 13 in app. C.4].

Fig. 9 (top) illustrates the memory consumption patterns for inline and RIARC monitoring, which exhibit *elasticity*. This elastic behaviour occurs at different points in the plots. Inline monitoring peaks at $\approx 3.7GB$ at $\approx 72s$ and RIARC at $\approx 5.7GB$ at $\approx 100s$ under the Burst workload. The memory consumption for both methods stabilises at around $\approx 36s$ under the Steady workload, with $\approx 2.3GB$ for inline and $\approx 2.7GB$ for RIARC monitoring.

Elasticity in these methods is due to different reasons: it is intrinsic to inline monitoring (see sec. 1), whereas the RIARC spawns and garbage collects monitors on demand (secs. 3.1 and 3.6). These observations are certified by [8, fig. 16 in app. C.6.3] under the Pulse workload. Centralised monitoring is *insensitive* to the workload applied, as [8, figs. 17 and 18 in app. C.6.3] reconfirm.

The effect of dynamic message routing and tracer reconfiguration that RIARC performs is evident in the scheduler utilisation plots of fig. 9. Under the Steady and Burst workloads, scheduler utilisation oscillates continually due to the sustained influx of trace events. Oscillations corroborate our observation in sec. 5.4.2 about RIARC, namely, that monitors are activated by trace events but remain idle otherwise. Active monitor periods manifest as peaks in fig. 9. Idle periods, where monitors are placed in the EVM waiting queues, are reflected as regions with low and stable scheduler utilisation. These oscillations showcase the message-driven aspect of RIARC, which analyses events asynchronously. Inlining exhibits minimal scheduler utilisation oscillations due to its lock-step execution with the SuS.

5.5 Moderate concurrency benchmarks

Our last experiment studies moderate concurrency scenarios C_M . The general-case platform P_G sets $n = 5k$ workers and $w = 10k$ requests per worker, and uses 16 EVM schedulers. We show that under these loads, RIARC induces overhead on par with inline monitoring.

Moderate concurrency alters the execution of the master-worker model, compared to our benchmarks of secs. 5.4.1–5.4.3. In this set-up, the master creates most of its worker processes at the initial stage of benchmark runs and spends the remaining time allocating work requests. This change grows the request throughput, e.g. see [8, tbl. 5 in app. C.4]. One consequence is that centralised monitoring consistently crashes under the rapid accumulation of messages in its mailbox. We, thus, limit our study to inline and RIARC monitoring.

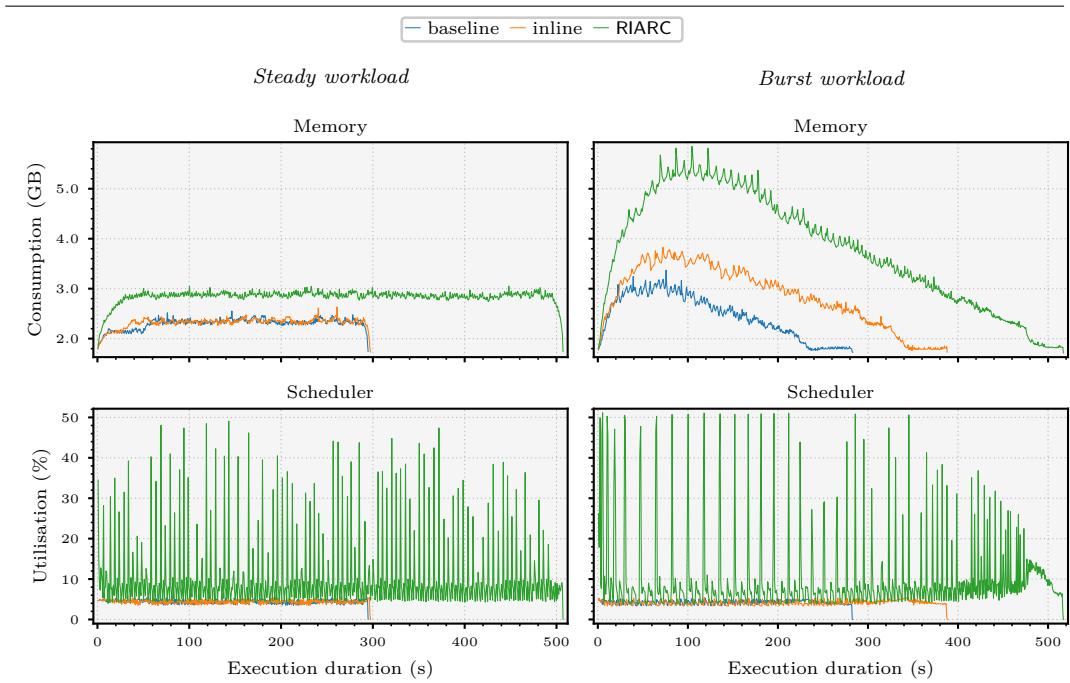


Figure 9 Inline and RIARC monitoring resource usage (*high* workload, 500k workers).

Tbl. 3 compares the results taken on platform P_G from sec. 5.4.3 with 500k workers (high concurrency, C_H) against the ones on P_G with 5k workers (moderate concurrency, C_M). The figures shown estimate the percentage overhead w.r.t. the baseline systems C_H and C_M at this *maximum* load. Our ensuing discussion is limited to the overhead under the Steady and Burst workloads since each respectively captures the SuS operation in *typical* and *severe* load conditions. Readers are referred to [8, fig. 20 in app. C.6.4] for the overhead comparison given in absolute metric values for the entirety of benchmark runs.

Tbl. 3 indicates that the memory consumption overhead due to inline monitoring is not affected under the Steady workload, which remains at 1% in both the high and moderate concurrency scenarios C_H and C_M . However, it decreases from 16% in C_H to 1% in C_M . We observe the opposite effect on the scheduler utilisation overhead for inline monitoring. For the moderate concurrency case C_M , the scheduler overhead under the Steady and Burst workloads increases to 3% and 4% respectively.

Tbl. 3 also shows that under the Steady workload, RIARC induces a 23% memory overhead in concurrency scenario C_H vs. 8% in concurrency scenario C_M , a decrease of 15%. Under the Burst workload, this overhead is reduced by 46%, from 56% in C_H to 10% in C_M . The scheduler utilisation overhead for RIARC from C_H to C_M also registers drops of $\approx 71\%$ under both Steady and Burst workloads. We attribute these overhead improvements to the lower number of worker processes the master creates in the moderate concurrency set-up, C_M . The long-running worker processes induce stability in the SuS. RIARC adapts to this change favourably by performing fewer trace event routing and tracer reconfigurations. The ramification of this adaptability is perceivable in the latency overhead discussed next.

RIARC inflates the latency overhead from 95% in C_H to 194% in C_M under the Steady workload (+99%), and from 97% in C_H to 190% in C_M under the Burst workload (+93%). However, RIARC induces *less latency* overhead than inline monitoring. Tbl. 3 reveals that the latency overhead for inline monitoring grows from 4% in the high concurrency set-up C_H to 246% in the moderate concurrency set-up C_M under the Steady workload (+242%). It also grows under the Burst workload, from 55% in C_H to 193% in C_M (+138%). In fact, our calculations confirm that the *absolute* response time for inline monitoring is slightly worse than that of RIARC in C_M : 116ms vs. 98ms under the Steady, and 182ms vs. 179ms under the Burst workloads respectively. This latency degradation for inline monitoring stems from the $\approx 5\mu s$ slowdown induced by the RV analysis, which results in frequent “pausing” of worker processes. Monitors comprising richer analyses produce longer pauses in worker processes, which can degrade the response time further [46, 37, 69].

■ **Table 3** Percentage overhead on C_H (500k) and C_M (5k) w.r.t. baseline at *maximum* workload.

Concurrency	Workload	Response time %	Memory consumption %	Scheduler utilisation %	
		Inline	RIARC	Inline	RIARC
C_H (500k)	Steady	4	95	1	23
	Burst	55	97	16	56
C_M (5k)	Steady	246	194	1	8
	Burst	193	190	1	10
				0	0
				3	3
				4	4
				123	123
				52	52
				50	50

5.6 Discussion

The RIARC scheduler utilisation in tbl. 3 is higher than the reported values for inline monitoring. This should not be construed as an inefficiency. From a reactive systems perspective, growth in the scheduler utilisation indicates *scalability*, as the low memory consumption in tbl. 3 affirms. RIARC benefits from the ample schedulers to improve the overall system response time *without* overtaxing the system. Indeed, [8, fig. 20 in app. C.6.4] demonstrates that the mean absolute scheduler utilisation in the benchmarks of sec. 5.5 is just $\approx 10\%$ under both the Steady and Burst workloads. Tbl. 3 shows that the reduction in latency makes RIARC comparable to inline monitoring in moderate concurrency scenarios.

Sec. 1 names *responsiveness* as a key reactive systems attribute [94]. RIARC prioritises responsiveness by isolating its monitors into asynchronous concurrent units. This design naturally exploits the available processing capacity of the host platform by maximising monitor *parallelism* when possible. Inline monitoring reaps fewer benefits in identical settings because its lock-step execution with the SuS robs it of potential parallelism gains.

Secs. 5.4.1 – 5.4.3 attest to the impracticality of centralised monitoring for reactive systems. Bottlenecks hinder its ability to scale, compelling it to consume inordinate amounts of memory, which can lead to failure, as sec. 5.4.2 shows. Despite these shortcomings, many RV tools in this setting use centralised monitoring, e.g. [50, 18, 126, 65, 81, 110, 72, 37, 41, 38, 2, 103].

6 Conclusion

Reactive software calls for instrumentation methods that uphold the responsive, resilient, message-driven, and elastic attributes of systems. This is attainable *only if* the instrumentation exhibits these qualities. Runtime verification imposes another demand on the instrumentation: the trace event sequences it reports to monitors must be *sound*, i.e., traces do not omit events and preserve the ordering with which events occur locally at processes.

This paper presents RIARC, a novel decentralised instrumentation algorithm for outline monitors meeting these two demands. RIARC uses outline monitors to decouple the runtime analysis from system components, which minimises latency and promotes *responsiveness*. Outline monitors can fail independently of the system and each other to improve *resiliency*. RIARC gathers events non-invasively via a tracing infrastructure, making it *message-driven* and suited to cases where inlining is inapplicable. The algorithm is *elastic*: it reacts to specific events in the trace to instrument and garbage collect monitors on demand.

Our asynchronous setting complicates the instrumentation due to potential trace event loss or reordering. RIARC overcomes these challenges using a next-hop IP routing approach to rearrange and report events soundly to monitors. We validate RIARC by subjecting its corresponding Erlang implementation to rigorous systematic testing, confirming its correctness. This implementation is validated via extensive empirical experiments. These subject the implementation to large realistic workloads to ascertain its reactiveness. Our experiments show that RIARC optimises its memory and scheduler usage to maintain latency feasible for soft real-time applications. We also compare RIARC to inline and centralised monitoring, revealing that it induces *comparable* latency to inlining under moderate concurrency.

Related work. Other work on inlining besides that cited in sec. 1, e.g. [78, 25, 50, 49, 53], does not separate the instrumentation and runtime analysis. This view is commonplace in monolithic settings, where the instrumentation is often assumed to induce minimal runtime overhead. As a result, many inline approaches focus on the efficiency of the analysis but neglect the instrumentation cost (e.g. [63] attributes overhead solely to the analysis). These

arguments for monolithic systems are often ported to concurrent settings. For instance, [107, 126, 29, 46, 125, 66, 21] propose efficient runtime monitoring algorithms but do not account for, nor quantify, the overhead due to gathering trace events. Tools that measure the runtime overhead, such as [41, 37, 19, 34, 72, 133], coalesce the instrumentation and runtime analysis costs, making it difficult to gauge the source of inefficiencies. Some literature [39, 52] even extends the assumption about minimal instrumentation overhead to offline monitoring, stating that the instrumentation consists of “only” capturing trace events. Sec. 5.4.1 shows this *not* to be the case. We are unaware of empirical studies such as ours that concretely distinguish between and quantify the instrumentation and runtime analysis overhead.

Sec. 5.6 remarks that centralised monitoring is used for concurrent runtime verification despite its evident limitations. One plausible reason for this is that the empirical scrutiny of such tools lacks proper benchmarking (e.g. [50, 18, 126, 65, 81]) or uses insufficient workloads that fail to expose the issues of centralised set-ups (e.g. [110, 72, 37, 41, 38, 2, 103]). Gathering inadequate metrics can also bias the interpretation of empirical data; see sec. 5.4.1. Works, such as [38, 19, 34, 124], consider the memory consumption and latency metrics. Our evaluation of inline, centralised, and RIARC monitoring uses (i) *combinations* of hardware and software, with (ii) two concurrency models that test *edge-case* and *general-case* scenarios, under (iii) *high* workloads that go beyond the state of the art, applying (iv) *realistic* workload profiles, interpreted against (v) *relevant* performance metrics that give a multi-faceted view of runtime overhead. To the best of our knowledge, this is generally not done in other studies, e.g. [114, 113, 47, 46, 119, 30, 106, 38, 41, 19, 50, 51, 53, 72, 59, 60, 27, 110, 97, 34].

Outline instrumentation decouples the execution of the SuS and monitor components in space (i.e., isolated threads) and time (i.e., asynchronous messaging). The tracing infrastructure outline instrumentation uses mirrors the publish-subscribe (Pub/Sub) pattern [129]. In this set-up, consumers subscribe to a *broker* that advertises events. Centralised instrumentation follows a Pub/Sub approach: the SuS produces trace events and deposits them into *one* global trace buffer that tracers receive from (see fig. 1b). Despite similarities, e.g. tracers register and deregister with the tracing infrastructure at runtime, RIARC differs from conventional Pub/Sub messaging in three fundamental aspects. Chiefly, Pub/Sub publishers are unaware of the subscribers interested in receiving messages because this bookkeeping task is appointed to the broker. By contrast, next-hop routing relies on knowing the *explicit* address of recipients to forward messages. Furthermore, in Pub/Sub messaging, subscribers do not communicate with publishers, whereas RIARC tracers exchange *direct* detach requests between one another to reorganise the choreography (refer to sec. 3.4). Lastly, Pub/Sub brokers are typically predefined and remain fixed, while trace partitioning *reconfigures* the tracing topology, creating and destroying brokers in reaction to dynamic changes in SuS.

One assumption we make about process tracing is A₄, i.e., tracing gathers the spawn events of parent processes before all the events of child processes. While A₄ induces a partial order over trace events, it is *weaker* than happened-before causality [95], as the events gathered from sets of child SuS processes need not be causally ordered. Demanding the latter condition would entail additional computation on the part of the tracing infrastructure and could increase runtime overhead. Maintaining minimal overhead is critical to our instrumentation because it preserves the responsiveness attribute of reactive systems. Tracing assumption A₄ and the RIARC logic detailed in sec. 3 guarantee trace soundness (def. 1), which suffices for RV monitoring. Since our work targets soft real-time systems [94, 92] scoped in a reliable messaging setting (see sec. 1), we do not tackle the problem of ensuring time-bounded causally-ordered message delivery [20] nor implement exactly-once delivery semantics [83]. We will address these challenges in future extensions of this work.

References

- 1 Francisco Lopez-Sancho Abraham. *Akka in Action*. Manning, 2023.
- 2 Luca Aceto, Antonis Achilleos, Elli Anastasiadi, and Adrian Francalanza. Monitoring Hyperproperties with Circuits. In *FORTE*, volume 13273 of *LNCS*, pages 1–10, 2022.
- 3 Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Léo Exibard, Adrian Francalanza, and Anna Ingólfssdóttir. A Monitoring Tool for Linear-Time μ HML. In *COORDINATION*, volume 13271 of *LNCS*, pages 200–219, 2022.
- 4 Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Léo Exibard, Adrian Francalanza, and Anna Ingólfssdóttir. A Monitoring Tool for Linear-time μ hml. *Sci. Comput. Program.*, 232:103031, 2024.
- 5 Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfssdóttir, and Karoliina Lehtinen. Adventures in Monitorability: From Branching to Linear Time and Back Again. *PACMPL*, 3:52:1–52:29, 2019.
- 6 Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfssdóttir, and Karoliina Lehtinen. An Operational Guide to Monitorability with Applications to Regular Properties. *Softw. Syst. Model.*, 20:335–361, 2021.
- 7 Luca Aceto, Duncan Paul Attard, Adrian Francalanza, and Anna Ingólfssdóttir. On Benchmarking for Concurrent Runtime Verification. In *FASE*, volume 12649 of *LNCS*, pages 3–23, 2021.
- 8 Luca Aceto, Duncan Paul Attard, Adrian Francalanza, and Anna Ingólfssdóttir. Runtime Instrumentation for Reactive Components. *CoRR*, abs/2406.19904, 2024.
- 9 Luca Aceto, Anna Ingólfssdóttir, Kim Guldstrand Larsen, and Jiří Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, 2007.
- 10 Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A Foundation for Actor Computation. *JFP*, 7:1–72, 1997.
- 11 Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *AFIPS Spring Joint Computing Conference*, volume 30 of *AFIPS Conference Proceedings*, pages 483–485, 1967.
- 12 Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.
- 13 Joe Armstrong. Erlang. *Commun. ACM*, 53(9):68–75, 2010.
- 14 Stavros Aronis. *Effective Techniques for Stateless Model Checking*. PhD thesis, Uppsala University, Sweden, 2018.
- 15 Duncan Paul Attard. Runtime Instrumentation for Reactive Components (Artifact). Software, version 2.0. (visited on 2024-08-05). URL: <https://doi.org/10.5281/zenodo.10634182>.
- 16 Duncan Paul Attard, Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfssdóttir, and Karoliina Lehtinen. Better Late than Never or: Verifying Asynchronous Components at Runtime. In *FORTE*, volume 12719 of *LNCS*, pages 207–225, 2021.
- 17 Duncan Paul Attard, Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfssdóttir. Introduction to Runtime Verification. In *Behavioural Types: from Theory to Tools*, Automation, Control and Robotics, pages 49–76. River, 2017.
- 18 Duncan Paul Attard and Adrian Francalanza. A Monitoring Tool for a Branching-Time Logic. In *RV*, volume 10012 of *LNCS*, pages 473–481, 2016.
- 19 Duncan Paul Attard and Adrian Francalanza. Trace Partitioning and Local Monitoring for Asynchronous Components. In *SEFM*, volume 10469 of *LNCS*, pages 219–235, 2017.
- 20 Roberto Baldoni, Achour Mostéfaoui, and Michel Raynal. Causal Delivery of Messages with Real-Time Data in Unreliable Networks. *Real Time Syst.*, 10(3):245–262, 1996.
- 21 Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David E. Rydeheard. Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors. In *FM*, volume 7436 of *LNCS*, pages 68–84, 2012.

- 22 Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to Runtime Verification. In *Lectures on Runtime Verification*, volume 10457 of *LNCS*, pages 1–33. Springer, 2018.
- 23 Basho. Bench, 2017. URL: https://github.com/basho/basho_bench.
- 24 David A. Basin, Felix Klaedtke, and Eugen Zalinescu. Failure-Aware Runtime Verification of Distributed Systems. In *FSTTCS*, volume 45 of *LIPICS*, pages 590–603, 2015.
- 25 Andreas Bauer and Yliès Falcone. Decentralised LTL Monitoring. *FMSD*, 48:46–93, 2016.
- 26 André Bento, Jaime Correia, Ricardo Filipe, Filipe Araújo, and Jorge Cardoso. Automated Analysis of Distributed Tracing: Challenges and Research Directions. *J. Grid Comput.*, 19(1):9, 2021.
- 27 Shay Berkovich, Borzoo Bonakdarpour, and Sebastian Fischmeister. Runtime Verification with Minimal Intrusion through Parallelism. *FMSD*, 46:317–348, 2015.
- 28 Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA*, pages 169–190, 2006.
- 29 Eric Bodden. The Design and Implementation of Formal Monitoring Techniques. In *OOPSLA Companion*, pages 939–940, 2007.
- 30 Eric Bodden, Laurie J. Hendren, Patrick Lam, Ondřej Lhoták, and Nomair A. Naeem. Collaborative Runtime Verification with Tracematches. *J. Log. Comput.*, 20:707–723, 2010.
- 31 Borzoo Bonakdarpour, Pierre Fraigniaud, Sergio Rajsbaum, David A. Rosenblueth, and Corentin Travers. Decentralized Asynchronous Crash-Resilient Runtime Verification. In *CONCUR*, volume 59 of *LIPICS*, pages 16:1–16:15, 2016.
- 32 Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson. The Reactive Manifesto, 2014.
- 33 Jonas Bonér and Viktor Klang. Reactive Programming vs. Reactive Systems. Technical report, Lightbend Inc., 2016.
- 34 Christian Bartolo Burlò, Adrian Francalanza, and Alceste Scalas. On the Monitorability of Session Types, in Theory and Practice. In *ECOOP*, volume 194 of *LIPICS*, pages 20:1–20:30, 2021.
- 35 Rajkumar Buyya, James Broberg, and Andrzej M. Goscinski. *Cloud Computing: Principles and Paradigms*. Wiley-Blackwell, 2011.
- 36 Bryan Cantrill. Hidden in Plain Sight. *ACM Queue*, 4:26–36, 2006.
- 37 Ian Cassar and Adrian Francalanza. On Synchronous and Asynchronous Monitor Instrumentation for Actor-based Systems. In *FOCLASA*, volume 175 of *EPTCS*, pages 54–68, 2014.
- 38 Ian Cassar and Adrian Francalanza. On Implementing a Monitor-Oriented Programming Framework for Actor Systems. In *IFM*, volume 9681 of *LNCS*, pages 176–192, 2016.
- 39 Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfssdóttir. A Survey of Runtime Monitoring Instrumentation Techniques. In *PrePostiFM*, volume 254 of *EPTCS*, pages 15–28, 2017.
- 40 Ian Cassar, Adrian Francalanza, Duncan Paul Attard, Luca Aceto, and Anna Ingólfssdóttir. A Suite of Monitoring Tools for Erlang. In *RV-CuBES*, volume 3 of *Kalpa Publications in Computing*, pages 41–47, 2017.
- 41 Ian Cassar, Adrian Francalanza, and Simon Said. Improving Runtime Overheads for detectEr. In *FESCA*, volume 178 of *EPTCS*, pages 1–8, 2015.
- 42 Francesco Cesarini and Simon Thompson. *Erlang Programming: A Concurrent Approach to Software Development*. O'Reilly Media, 2009.
- 43 Bernadette Charron-Bost, Friedemann Mattern, and Gerard Tel. Synchronous, Asynchronous, and Causally Ordered Communication. *Distributed Comput.*, 9(4):173–191, 1996.

- 44 Natalia Chechina, Kenneth MacKenzie, Simon J. Thompson, Phil Trinder, Olivier Boudeville, Viktoria Fordós, Csaba Hoch, Amir Ghaffari, and Mario Moro Hernandez. Evaluating Scalable Distributed Erlang for Scalability and Reliability. *IEEE Trans. Parallel Distributed Syst.*, 28(8):2244–2257, 2017.
- 45 Feng Chen and Grigore Rosu. Java-MOP: A Monitoring Oriented Programming Environment for Java. In *TACAS*, volume 3440 of *LNCS*, pages 546–550, 2005.
- 46 Feng Chen and Grigore Rosu. Mop: An Efficient and Generic Runtime Verification Framework. In *OOPSLA*, pages 569–588, 2007.
- 47 Feng Chen and Grigore Rosu. Parametric Trace Slicing and Monitoring. In *TACAS*, volume 5505 of *LNCS*, pages 246–261, 2009.
- 48 Maria Christakis, Alkis Gotovos, and Konstantinos Sagonas. Systematic Testing for Detecting Concurrency Errors in Erlang Programs. In *ICST*, pages 154–163. IEEE Computer Society, 2013.
- 49 Christian Colombo and Yliès Falcone. Organising LTL Monitors over Distributed Systems with a Global Clock. *FMSD*, 49:109–158, 2016.
- 50 Christian Colombo, Adrian Francalanza, and Rudolph Gatt. Elarva: A Monitoring Tool for Erlang. In *RV*, volume 7186 of *LNCS*, pages 370–374, 2011.
- 51 Christian Colombo, Adrian Francalanza, Ruth Mizzi, and Gordon J. Pace. polyLarva: Runtime Verification with Configurable Resource-Aware Monitoring Boundaries. In *SEFM*, volume 7504 of *LNCS*, pages 218–232, 2012.
- 52 Christian Colombo and Gordon J. Pace. *Runtime Verification - A Hands-On Approach in Java*. Springer, 2022.
- 53 Christian Colombo, Gordon J. Pace, and Gerardo Schneider. LARVA — Safer Monitoring of Real-Time Java Programs (Tool Paper). In *SEFM*, pages 33–37, 2009.
- 54 Markus Dahm. Byte Code Engineering with the BCEL API. Technical report, Java Informationstage 99, 2001.
- 55 Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51:107–113, 2008.
- 56 Mathieu Desnoyers and Michel Dagenais. The LTng Tracer: A Low Impact Performance and Behavior Monitor for GNU/Linux. Technical report, École Polytechnique de Montréal, 2006.
- 57 Jean Dollimore, Tim Kindberg, and George Coulouris. *Distributed Systems: Concepts and Design*. Addison-Wesley, 2005.
- 58 Eclipse/IBM. OpenJ9, 2021. URL: <https://www.eclipse.org/openj9>.
- 59 Antoine El-Hokayem and Yliès Falcone. Monitoring Decentralized Specifications. In *ISSTA*, pages 125–135, 2017.
- 60 Antoine El-Hokayem and Yliès Falcone. On the Monitoring of Decentralized Specifications: Semantics, Properties, Analysis, and Simulation. *ACM Trans. Softw. Eng. Methodol.*, 29:1:1–1:57, 2020.
- 61 Úlfar Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, US, 2004.
- 62 Úlfar Erlingsson and Fred B. Schneider. SASI Enforcement of Security Policies: A Retrospective. In *NSPW*, pages 87–95, 1999.
- 63 Yliès Falcone, Klaus Havelund, and Giles Reger. A Tutorial on Runtime Verification. In *Engineering Dependable Software Systems*, volume 34 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 141–175. IOS Press, 2013.
- 64 Yliès Falcone, Srđan Krstić, Giles Reger, and Dmitriy Traytel. A Taxonomy for Classifying Runtime Verification Tools. *STTT*, 23:255–284, 2021.
- 65 Yliès Falcone, Hosein Nazarpour, Saddek Bensalem, and Marius Bozga. Monitoring Distributed Component-Based Systems. In *FACS*, volume 13077 of *LNCS*, pages 153–173, 2021.
- 66 Yliès Falcone, Hosein Nazarpour, Mohamad Jaber, Marius Bozga, and Saddek Bensalem. Tracing Distributed Component-Based Systems, a Brief Overview. In *RV*, volume 11237 of *LNCS*, pages 417–425, 2018.

- 67 Apache Software Foundation. JMeter, 2020. URL: <https://jmeter.apache.org>.
- 68 Pierre Fraigniaud, Sergio Rajsbaum, and Corentin Travers. On the Number of Opinions Needed for Fault-Tolerant Run-Time Monitoring in Distributed Systems. In *RV*, volume 8734 of *LNCS*, pages 92–107, 2014.
- 69 Adrian Francalanza. A Theory of Monitors. *Inf. Comput.*, 281:104704, 2021.
- 70 Adrian Francalanza, Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Ian Cassar, Dario Della Monica, and Anna Ingólfssdóttir. A Foundation for Runtime Monitoring. In *RV*, volume 10548 of *LNCS*, pages 8–29, 2017.
- 71 Adrian Francalanza, Jorge A. Pérez, and César Sánchez. Runtime Verification for Decentralised and Distributed Systems. In *Lectures on RV*, volume 10457 of *LNCS*, pages 176–210. Springer, 2018.
- 72 Adrian Francalanza and Aldrin Seychell. Synthesising Correct Concurrent Runtime Monitors. *FMSD*, 46:226–261, 2015.
- 73 Sukumar Ghosh. *Distributed Systems: An Algorithmic Approach*. CRC, 2014.
- 74 Patrice Godefroid. Model Checking for Programming Languages using Verisoft. In *POPL*, pages 174–186. ACM Press, 1997.
- 75 Susanne Graf, Doron A. Peled, and Sophie Quinton. Monitoring Distributed Systems Using Knowledge. In *FORTE*, volume 6722 of *LNCS*, pages 183–197, 2011.
- 76 Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: A Call Graph Execution Profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126. ACM, 1982.
- 77 Jim Gray. *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann, 1993.
- 78 Radu Grigore, Dino Distefano, Rasmus Lerchedahl Petersen, and Nikos Tzevelekos. Runtime Verification Based on Register Automata. In *TACAS*, volume 7795 of *LNCS*, pages 260–276, 2013.
- 79 Duncan A. Grove and Paul D. Coddington. Analytical Models of Probability Distributions for MPI Point-to-Point Communication Times on Distributed Memory Parallel Computers. In *ICA3PP*, volume 3719 of *LNCS*, pages 406–415, 2005.
- 80 Eric A. Hall. *Internet Core Protocols: The Definitive Guide*. O'Reilly Media, 2000.
- 81 Klaus Havelund, Giles Reger, Daniel Thoma, and Eugen Zalinescu. Monitoring Events that Carry Data. In *Lectures on Runtime Verification*, volume 10457 of *LNCS*, pages 61–102. Springer, 2018.
- 82 Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI*, pages 235–245, 1973.
- 83 Yongqiang Huang and Hector Garcia-Molina. Exactly-Once Semantics in a Replicated Messaging System. In *ICDE*, pages 3–12. IEEE Computer Society, 2001.
- 84 Shams Mahmood Imam and Vivek Sarkar. Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. In *AGERE!@SPLASH*, pages 67–80, 2014.
- 85 Justin Iurman, Frank Brockners, and Benoit Donnet. Towards Cross-Layer Telemetry. In *ANRW*, pages 15–21. ACM, 2021.
- 86 Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. CRC, 2020.
- 87 Nicolai M. Josuttis. *SOA in Practice: The Art of Distributed System Design: Theory in Practice*. O'Reilly Media, 2007.
- 88 Saša Jurić. *Elixir in Action*. Manning, 2019.
- 89 Bill Kayser. What is the expected distribution of website response times?, 2017. URL: <https://blog.newrelic.com/engineering/expected-distributions-website-response-times>.
- 90 Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *ECOOP*, volume 2072 of *LNCS*, pages 327–353, 2001.
- 91 Moonzoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-MaC: A Run-Time Assurance Approach for Java Programs. *FMSD*, 24:129–155, 2004.

- 92 Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications (Real-Time Systems Series)*. Springer, 2011.
- 93 Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2011.
- 94 Roland Kuhn, Brian Hanafee, and Jamie Allen. *Reactive Design Patterns*. Manning, 2016.
- 95 Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978.
- 96 Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4:382–401, 1982.
- 97 Julien Lange and Nobuko Yoshida. Verifying Asynchronous Interactions via Communicating Session Automata. In *CAV*, volume 11561 of *LNCS*, pages 97–117, 2019.
- 98 Paul Lavery and Takuo Watanabe. An Actor-Based Runtime Monitoring System for Web and Desktop Applications. In *SNPD*, pages 385–390. IEEE Computer Society, 2017.
- 99 Philipp Lengauer, Verena Bitto, Hanspeter Mössenböck, and Markus Weninger. A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008. In *ICPE*, pages 3–14, 2017.
- 100 Bryon C. Lewis and Albert E. Crews. The Evolution of Benchmarking as a Computer Performance Evaluation Technique. *MIS Q.*, 9:7–16, 1985.
- 101 Jay Ligatti, Lujo Bauer, and David Walker. Edit Automata: Enforcement Mechanisms for Run-Time Security Policies. *Int. J. Inf. Sec.*, 4:2–16, 2005.
- 102 Zhen Liu, Nicolas Niclausse, and César Jalpa-Villanueva. Traffic Model and Performance Evaluation of Web Servers. *Perform. Evaluation*, 46:77–100, 2001.
- 103 Qingzhou Luo and Grigore Rosu. EnforceMOP: A Runtime Property Enforcement System for Multithreaded Programs. In *ISSTA*, pages 156–166, 2013.
- 104 Deep Medhi and Karthik Ramasamy. Chapter 3 - routing protocols: Framework and principles. In *Network Routing (Second Edition)*, The Morgan Kaufmann Series in Networking, pages 64–113. Morgan Kaufmann, 2018.
- 105 Silvana M. Melo, Jeffrey C. Carver, Paulo S. L. Souza, and Simone R. S. Souza. Empirical Research on Concurrent Software Testing: A Systematic Mapping Study. *Inf. Softw. Technol.*, 105:226–251, 2019.
- 106 Patrick O’Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Rosu. An Overview of the MOP Runtime Verification Framework. *STTT*, 14:249–289, 2012.
- 107 Patrick O’Neil Meredith and Grigore Rosu. Efficient Parametric Runtime Verification with Deterministic String Rewriting. In *ASE*, pages 70–80, 2013.
- 108 Microsoft. MSDN, 2021. URL: <https://msdn.microsoft.com>.
- 109 Ian Molyneaux. *The Art of Application Performance Testing 2e*. O’Reilly Media, 2014.
- 110 Menna Mostafa and Borzoo Bonakdarpour. Decentralized Runtime Verification of LTL Specifications in Distributed Systems. In *IPDPS*, pages 494–503, 2015.
- 111 Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *PLDI*, pages 89–100. ACM, 2007.
- 112 Rumyana Neykova. *Multiparty Session Types for Dynamic Verification of Distributed Systems*. PhD thesis, Imperial College London, UK, 2017.
- 113 Rumyana Neykova and Nobuko Yoshida. Let it Recover: Multiparty Protocol-Induced Recovery. In *CC*, pages 98–108, 2017.
- 114 Rumyana Neykova and Nobuko Yoshida. Multiparty Session Actors. *LMCS*, 13, 2017.
- 115 Nicolas Niclausse. Tsung, 2017. URL: <http://tsung.erlang-projects.org>.
- 116 Scott Oaks. *Java Performance: In-Depth Advice for Tuning and Programming Java 8, 11, and Beyond*. CRC, 2020.
- 117 Martin Odersky, Lex Spoon, Bill Venners, and Frank Sommers. *Programming in Scala*. Artima Inc., 2021.
- 118 Kevin Quick. Thespian, 2020. URL: <https://thespianpy.com/doc>.

- 119 Giles Reger, Helena Cuenca Cruz, and David E. Rydeheard. MarQ: Monitoring at Runtime with QEA. In *TACAS*, volume 9035 of *LNCS*, pages 596–610, 2015.
- 120 Sartaj Sahni and George L. Vairaktarakis. The Master-Slave Paradigm in Parallel Computer and Industrial Settings. *J. Glob. Optim.*, 9:357–377, 1996.
- 121 Raja R. Sambasivan, Ilari Shafer, Jonathan Mace, Benjamin H. Sigelman, Rodrigo Fonseca, and Gregory R. Ganger. Principled Workflow-Centric Tracing of Distributed Systems. In *SoCC*, pages 401–414. ACM, 2016.
- 122 Torben Scheffel and Malte Schmitz. Three-Valued Asynchronous Distributed Runtime Verification. In *MEMOCODE*, pages 52–61, 2014.
- 123 Fred B. Schneider. Enforceable Security Policies. *ACM Trans. Inf. Syst. Secur.*, 3:30–50, 2000.
- 124 Joshua Schneider, David A. Basin, Frederik Brix, Srdan Krstic, and Dmitriy Traytel. Scalable Online First-Order Monitoring. *Int. J. Softw. Tools Technol. Transf.*, 23:185–208, 2021.
- 125 Koushik Sen, Grigore Rosu, and Gul Agha. Runtime Safety Analysis of Multithreaded Programs. In *ESEC / SIGSOFT FSE*, pages 337–346, 2003.
- 126 Koushik Sen, Grigore Rosu, and Gul Agha. Online Efficient Predictive Safety Analysis of Multithreaded Programs. *Int. J. Softw. Tools Technol. Transf.*, 8:248–260, 2006.
- 127 Koushik Sen, Abhay Vardhan, Gul Agha, and Grigore Rosu. Efficient Decentralized Monitoring of Safety in Distributed Systems. In *ICSE*, pages 418–427, 2004.
- 128 Eric Stenman. The erlang runtime system, 2023.
- 129 Sasu Tarkoma. *Overlay Networks: Toward Information Networking*. Auerbach, 2010.
- 130 The Pony Team. Ponylang, 2021. URL: <https://tutorial.ponylang.io>.
- 131 Ulf T. Wiger, Gösta Ask, and Kent Boortz. World-Class Product Certification using Erlang. *ACM SIGPLAN Notices*, 37(12):25–34, 2002.
- 132 Jiali Yao, Zhigeng Pan, and Hongxin Zhang. A Distributed Render Farm System for Animation Production. In *ICEC*, volume 5709 of *LNCS*, pages 264–269, 2009.
- 133 Teng Zhang, Greg Eakman, Insup Lee, and Oleg Sokolsky. Overhead-Aware Deployment of Runtime Monitors. In *RV*, volume 11757 of *LNCS*, pages 375–381, 2019.

A Dynamic Logic for Symbolic Execution for the Smart Contract Programming Language Michelson

Barnabas Arvay 

University of Freiburg, Germany

Thi Thu Ha Doan 

University of Freiburg, Germany

Peter Thiemann 

University of Freiburg, Germany

Abstract

Verification of smart contracts is an important topic in the context of blockchain technology. We study an approach to verification that is based on symbolic execution.

As a formal basis for symbolic execution, we design a dynamic logic for Michelson, the smart contract language of the Tezos blockchain, and prove its soundness in the proof assistant Agda. Towards the soundness proof we formalize the concrete semantics as well as its symbolic counterpart in a unified setting. The logic encompasses single contract runs as well as inter-contract runs chained in a single transaction.

2012 ACM Subject Classification Software and its engineering → Automated static analysis

Keywords and phrases Smart Contract, Blockchain, Formal Verification, Symbolic Execution

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.3

Supplementary Material

Software (Source Code): <https://freidok.uni-freiburg.de/data/255176> [6]

Funding *Thi Thu Ha Doan:* Supported by the Tezos Foundation, grant COOC.

1 Introduction

Blockchain technology and smart contracts provide decentralized and immutable systems for secure transactions and automated agreements. Smart contracts have been targets of spectacular and costly attacks as contracts are immutable and their source code is publicly available on the blockchain. Hence, it is vital as well as challenging to ensure the correctness of smart contracts before their deployment. Formal methods and various verification techniques have been proposed to address this challenge.

The Tezos blockchain [14] and its smart contract language Michelson have been designed from ground up with verification in mind. Several frameworks have been developed based on, e.g., interactive theorem proving [10], refinement typing [27], and automated theorem proving [5]. We are interested in automated verification of Michelson programs, which rules out interactive approaches. Symbolic execution [20, 11] is one of the standard approaches to automatically obtain verification conditions like weakest preconditions for failures as well as normal termination from a program. Next, an SMT-solver discharges these verification conditions. There is a wide range of approaches that apply symbolic execution combined with SMT-solving to smart contracts, mostly for the Ethereum blockchain (see Section 6).

While there are many approaches to symbolic execution [12, 13, 30], we choose one based on dynamic logic. Dynamic logic (DL) [16] is a modal logic for reasoning about programs. Its signature features are modalities for program execution. These modalities enable the expression of assertions about program behavior as logical formulas. For instance, the formula

3:2 Dynamic Logic for Symbolic Execution for Michelson

$[p]\Psi$ states partial correctness: if program p terminates, then Ψ is true. That is, a Hoare triple $\{\Phi\} p \{\Psi\}$ can be encoded by $\Phi \rightarrow [p]\Psi$. DL also provides a modality $\langle p \rangle$ for total correctness, but we do not consider it in this work.

Dynamic logic comes with proof rules for the modality derived from the structure of p . For example, if $p; q$ stands for sequential execution of p and q , then the proof rule $[p; q]\Psi \leftrightarrow [p][q]\Psi$ states that execution of p enables execution of q such that Ψ holds in the end. Similarly, the rule $[\varepsilon]\Psi \leftrightarrow \Psi$ states that the empty program ε does not modify the validity of Ψ .

In the past, dynamic logic has been used successfully for as a basis for symbolic execution in the context of the verification of Java programs [9], as it is particularly well suited to keep track of a changing environment (i.e., mutable objects on Java’s heap). We design a DL to model Michelson execution because we want to reason about transactions that span several contract runs. In Michelson terminology, these transactions are called *chained contract executions*, where an externally started contract run initiates further internal contract runs. Our DL design models the relevant parts of the blockchain run-time system on top of the purely functional execution of Michelson programs. On the level of the run-time system contracts are very similar to objects: they are identified by an address and they come with mutable attributes (state and balance).

The DL treatment of the functional part of Michelson is quite intuitive: programs are sequences of Michelson instructions, we model the execution state of a Michelson program by a formula of the form $\Phi \rightarrow [p]\Psi$, and the proof rules for $[i;p]\Psi$ (where i is a single instruction) define the semantics of symbolic execution.

Gas is an important aspect of computation on the blockchain. The initial caller of a contract has to pay for executing the transaction (consisting of one or more chained contract runs) in terms of gas. A transaction that runs out of gas is rolled back by the run-time system of the blockchain as if it never happened. As Michelson does not suffer from reentrancy problems (cf. Section 2), gas does not affect reasoning about the functional correctness of (chained) contract execution. For that reason, our DL design does not account for gas.

It is the sole goal of this paper to provide a **machine verified specification** of symbolic execution for Michelson, rather than an efficient or otherwise realistic implementation. For that reason, the paper does not cover all instructions, but rather a carefully chosen representative subset. This is in contrast to related work [10, 27, 5] that describes **actual verification tools**. To be useful for a wide range of programs, such a tool must support as many Michelson instructions as possible¹, it must be reasonably efficient, and it must deal with loops and nontermination in an appropriate way. None of these issues are concerns for our specification.

Contributions

1. We select a representative subset of Michelson instructions so as to provide proof templates for all current and future instructions that work similarly.
2. We provide a parameterized semantics definition with instances for the concrete semantics as well as for an abstract semantics, which implements the dynamic logic for Michelson.
3. We prove the soundness of this logic first for single programs, and then for several programs chained in a transaction.

The Agda implementation of the contributions is available.²

¹ Keeping up with the rapid evolution of the language is challenging for those tools. As of this writing, most of them support the instruction set available in late 2022.

² <https://freidok.uni-freiburg.de/data/255176>, development version <https://github.com/Tezos-Project-Uni-Freiburg/michelson-dynamic-logic>.

Overview

Section 2 gives an overview of Michelson, introduces its type system and our intrinsically typed representation of the language. Section 3 defines the execution model of Michelson, first for single contracts, and then for the chained execution of several contracts that call each other. Section 4 introduces dynamic logic and its symbolic execution rules, again first for single execution, and then for chained execution. Section 5 explains the major components of the soundness proof of the dynamic logic. Section 6 discusses related work and conclusions.

The paper contains many excerpts from the live, type checked definitions and proofs in Agda. In particular, all major definitions and statements of theorems are shown in Agda notation to ensure consistency of the paper with the machine-checked proofs.

2 Michelson

Michelson [25, 28] is the native language for smart contracts on the Tezos blockchain. It is a low level, stack-based, simply-typed, purely functional programming language. That is, all computation is driven by transforming an input stack into an output stack. There are no mutable data structures; blockchain transactions are handled outside of Michelson. All contracts are statically typed to avoid run-time type errors.

Each Michelson instruction transforms a given input stack into an output stack where some of its values have been changed, added, or removed. For example, the ADD instruction accepts any stack whose two topmost elements are numbers, and returns a stack where these two values have been replaced by their sum. The remaining stack is unchanged.

`ADD : 15 :: 27 :: remainingStack \mapsto 42 :: remainingStack`

2.1 Types

Michelson supports the usual data types like numbers, pairs, and lists as well as some blockchain-specific types for tokens and contracts. Figure 1a contains Agda definitions for a select subset of Michelson types `Type`. As some base types can be treated alike later on, we represent them with a separate type `BaseType`.

Most types' names are self explanatory. The base type '`mutez`' stands for tokens, `addr` stands for blockchain addresses in Tezos. We introduce shorthand patterns for base types for readability. The type `operation` consists of blockchain operations that can be emitted during contract execution. This mechanism implements token transfers from the current contract to other accounts or contracts. The type `contract P` represents such a contract which accepts a parameter of type `ty` represented by `P : Passable ty`. The type predicate `Passable : Type \rightarrow Set` originates from the Michelson specification and characterizes types that can be passed as parameters to contracts. Its declaration is mutually recursive with `Type`.

The semantics of types is defined by a mapping to Agda types. Most Michelson types have obvious Agda counterparts, except `addr`, `contract`, and `operation`. Addresses and contracts are both represented by natural numbers. The difference is that a value of type `contract` is known to be a valid address of a contract of suitable type. We only define one alternative of the `Operation` datatype: `transfer-tokens v m c`, which models a token transfer to contract `c` while passing the parameter value `v` and tokens `m`.³

³ At the time of writing this paper, full Michelson also supports the operations `CREATE-CONTRACT`, `EMIT` (deliver an event to an external application), and `SET-DELEGATE` (delegate stakes to another account).

```

Addr = N - blockchain addresses
Mutez = N - Tezos currency
data Operation : Set

data BaseType : Set where
  'unit 'nat 'addr 'mutez : BaseType

data Type where
  operation : Type
  base      : BaseType → Type
  pair      : Type → Type → Type
  list option : Type → Type
  contract   : ∀ {t} → Passable t → Type

pattern unit  = base 'unit
pattern nat   = base 'nat
pattern addr  = base 'addr
pattern mutez = base 'mutez

[ ] : Type → Set
[ unit ]     = T
[ nat ]      = N
[ addr ]     = Addr
[ mutez ]    = Mutez
[ operation ] = Operation
[ pair t1 t2 ] = [ t1 ] × [ t2 ]
[ list t ]   = List [ t ]
[ option t ] = Maybe [ t ]
[ contract P ] = Addr

data Operation where
  transfer-tokens : ∀ {P : Passable t}
    → [ t ] → [ mutez ] → [ contract P ]
    → Operation

```

(a) Syntax.

(b) Semantics.

Figure 1 Michelson Types.

2.2 Programs and Instructions

Michelson programs are intrinsically typed, that is, only well-typed programs can be written. Accordingly, they are represented in Agda by a datatype `Program` indexed by the types on the input stack and the types on the output stack. We assume that `Stack` = `List Type`.

```

data Program : Stack → Stack → Set
data Instruction : Stack → Stack → Set

data Program where
  end : Program S S
  _ ;_ : Instruction Si So → Program So Se → Program Si Se

```

Instructions are indexed in the same way: If instruction `inst` maps an input stack of type `Si` to an output stack of type `So` and `prg` maps that output stack `So` to the final stack of type `Se`, then `inst ; prg` is a program that maps `Si` to `Se`. The empty program `end` does not transform the stack.

We discuss a representative subset of Michelson instructions shown in Figure 2. The definition of `Instruction+` implements the pattern that most instructions only transform a fixed number of elements on top of the input stack and are parametric in the rest.

The first group of instructions operates on a fixed number of values on the stack and pushes the result. All arithmetic operations belong to this group and we just give two examples, `ADDnn` and `ADDm`, which perform addition of natural numbers and tokens, respectively. Michelson language overloads arithmetic operators, but as overloading is not supported by Agda, we supply separate instructions. We come back to this issue at the end of this section.

```

Instruction+ : Stack → Stack → Set
Instruction+ a b = ∀ {s} → Instruction (a ++ s) (b ++ s)

data Instruction where
  ADDnn : Instruction+ [ nat ; nat ] [ nat ]
  ADDm : Instruction+ [ mutez ; mutez ] [ mutez ]
  CAR : Instruction+ [ pair t1 t2 ] [ t1 ]
  CDR : Instruction+ [ pair t1 t2 ] [ t2 ]
  PAIR : Instruction+ [ t1 ; t2 ] [ pair t1 t2 ]
  NONE : ∀ t → Instruction+ [] [ option t ]
  SOME : Instruction+ [ t ] [ option t ]
  NIL : ∀ t → Instruction+ [] [ list t ]
  CONS : Instruction+ [ t ; list t ] [ list t ]
  TRANSFER-TOKENS : ∀ {P : Passable t} → Instruction+ [ t ; mutez ; contract P ] [ operation ]

  DROP : Instruction+ [ t ] []
  DUP : Instruction+ [ t ] [ t ; t ]
  SWAP : Instruction+ [ t1 ; t2 ] [ t2 ; t1 ]
  UNPAIR : Instruction+ [ pair t1 t2 ] [ t1 ; t2 ]

  AMOUNT : Instruction+ [] [ mutez ]
  BALANCE : Instruction+ [] [ mutez ]
  CONTRACT : (P : Passable t) → Instruction+ [ addr ] [ option (contract P) ]

  PUSH : Data t → Instruction+ [] [ t ]

  IF-NONE : Program S Se → Program (t :: S) Se → Instruction (option t :: S) Se
  ITER : Program (t :: S) S → Instruction (list t :: S) S
  DIP : ∀ n → {T (n ≤b length S)} → Program (drop n S) Se → Instruction S (take n S ++ Se)

```

■ **Figure 2** Instructions of Core Michelson.

CAR, **CDR**, and **PAIR** are the standard operations on pairs. **NONE** and **SOME** are the constructors for the **option** datatype, and **NIL** and **CONS** construct lists. The constructors for “empty” containers, **NONE** and **NIL** are indexed by the element type, otherwise that type can be inferred from the context.

The last instruction in this group is **TRANSFER-TOKENS**. Despite the name, this instruction does **not** directly transfer tokens to another account. It rather constructs a value **transfer-tokens** *v m c* of type **operation** from its arguments.

The instructions in the next group differ in that they push zero or more values on the output stack. **DROP** pops the stack, **DUP** duplicates the top of the stack, **SWAP** swaps the top entries, and **UNPAIR** eliminates a pair and pushes its contents. **UNPAIR** is a convenience instruction as it is equivalent to the instruction sequence **DUP**; **CDR**; **SWAP**; **CAR**.

The next group contains instructions that are blockchain specific. **AMOUNT** returns the tokens that were transferred with the currently running contract invocation and **BALANCE** returns the tokens currently owned by it. The **CONTRACT** instruction is indexed by a type *t* that must be **Passable**. It takes an address and checks on the blockchain whether this address is associated with a contract that accepts arguments of type *t*. The result is communicated as an **option** type. That is, the **contract** type carries a verified address.

The **PUSH** instruction pushes a value of type *t* on the stack. The value is encoded by a type-indexed datatype **Data** for pushable values. We elide its straightforward definition.

3:6 Dynamic Logic for Symbolic Execution for Michelson

The last group of instructions showcases control structures and an instruction that operates in a non-uniform way on the stack. The instruction **IF-NONE** eliminates a value of **option** type from the top of the stack. Its parameters are programs that implement the branches for case **None** and **Some**. The latter takes the value wrapped in the **Some** constructor as an argument on top of the stack.

The instruction **ITER** runs a sub-program on every element of its argument list. The instruction **DIP** n runs a sub-program at depth n on the input stack, that is, it skips over the first n elements of the stack, runs the sub-program, and reattaches those elements. The extra machinery in the implicit argument of the instruction makes sure that there are at least n elements on the stack. This mechanism is called reflection in the PLFA textbook [33].

Earlier, we remarked that Agda does not allow overloading of constructors in the same datatype. However, we can use reflection to define a “smart constructor” that almost suits the purpose.

```

overADD : (t1 t2 : Type) → Maybe (Ξ[ t ] Instruction+ [ t1 ; t2 ] [ t ])
overADD nat   nat   = just (nat   , ADDnn)
overADD mutez mutez = just (mutez , ADDm)
overADD _      _     = nothing

ADD : ∀ (p : map proj1 (overADD t1 t2) ≡ just t) → Instruction+ [ t1 ; t2 ] [ t ]
ADD{t1} {t2} {t} p with overADD t1 t2
... | just (t , add) with just-injective p
... | refl = add

```

The definition exploits the fact that the input stack of an instruction is always known in a Michelson program. The same fact also enables overloading in Michelson’s implementation to work. The function **overADD** specifies the resolution of overloading for the **ADD** instruction. If the argument types are both **nat**, then the result type is **nat** and the chosen instruction is **ADDnn**; and so on.⁴ If no overloading is known for a combination of arguments, the function returns **nothing**. The smart constructor **ADD** takes a proof that the overloading is defined on a given pair of input types. Then it extracts the selected instruction from the overloading.

Compared to “real” Michelson, the smart constructor requires an extra argument to work:

```

exnat : Program [ (pair nat nat) ] [ (pair nat nat) ]
exnat = DUP ; UNPAIR ; ADD refl ; DROP ; end

```

2.3 Blockchain Interface

A contract on the Tezos blockchain is indexed by a parameter type p and a store type s . The type p must be **Passable** and the type s must be **Storable**. Moreover, each contract comes with a current balance of tokens and a store of type s . The implementation of the contract is a program that maps a **pair** $p\ s$ to a **pair** (**list operation**) s , that is, it consumes the parameter paired with the current store and produces a list of operations (e.g., to invoke further contracts) paired with the updated store. The program itself is pure; any side effects, i.e., store update and contract calls, are managed by the blockchain runtime.

⁴ The full Michelson language has ten different overloadings of **ADD**.

```
record Contract (Mode : MODE) (p s : Type) : Set where
  constructor ctr
  field Param  : Passable p
  Store     : Storable s
  balance   : M Mode mutez
  storage   : M Mode s
  program   : Program [ pair p s ] [ pair (list operation) s ]
```

The *Mode* argument abstracts over the semantics of types. Its type has three components, one \mathcal{M} for the semantics and the others, \mathcal{F} and \mathcal{G} , are used by the abstract semantics in Subsection 4.2.

```
record MODE : Set1 where
  field M : Type → Set ; F : Set ; G : Set
```

Its instantiation for the concrete semantics installs the standard semantics of types from Section 2.1. The remaining components are instantiated to the unit type \top .

```
CMode : MODE
CMode = record { M = [] ; F = ⊤ ; G = ⊤ }
```

With this definition, the contract store of the blockchain is just a partial mapping from addresses to contracts.

```
Blockchain : (Mode : MODE) → Set
Blockchain Mode = Addr → Maybe (Ξ[ p ] Ξ[ s ] Contract Mode p s)
```

To start executing a contract, we initiate a blockchain transaction to its address, i.e., we ask the blockchain runtime to transfer tokens to its address along with its parameter. Once a contract has terminated, the runtime updates the stored value and processes the list of operations.

On the Tezos blockchain a normal account with deposit *init* corresponds to a contract with a unit parameter, unit store, and a trivial program that issues no operations.

```
Account : Mutez → Contract CMode unit unit
Account init = ctr unit unit init tt (CDR ; NIL operation ; PAIR ; end)
```

3 Michelson Reference Implementation

Program execution is defined in a small-step manner by a function that maps the current execution state of a program to a new state resulting from executing the first instruction:

```
prog-step : CProgState ro → CProgState ro
```

The type `CProgState ro` is a record that contains an input stack type ri , a program that maps an ri stack to an ro stack, an input stack of type ri , and the execution environment. `prog-step` executes the first instruction that must map an ri stack to an intermediate stack of type re , say. Consequently, the program in the output `CProgState` maps an re stack to an ro stack. As instructions as well as programs are intrinsically typed, the intermediate stack type re is sure to match. Likewise, the typing of `prog-step` ensures type preservation.

```

record ProgState (Mode : MODE) (ro : Stack) : Set where
  constructor state
  field {ri} : Stack
    en : Environment Mode
    prg : ShadowProg{M Mode} ri ro
    stk : All (M Mode) ri
    Φ : F Mode

prog-step ρ | fct ft ; p
= record ρ { prg = p ; stk = app-fct ft (H.front (stk ρ)) H.++ H.rest (stk ρ) }
prog-step ρ | DROP ; p
= record ρ { prg = p ; stk = H.rest (stk ρ) }

```

■ **Figure 3** Program state and single program step execution (excerpt).

3.1 Program Execution

So far we only concerned ourselves with the type of a Michelson stack. For program execution, both the types and values of stack elements are relevant. To this end, we have to lift the interpretation of a single type, i.e., a function from `Type` to `Set`, to the interpretation of a list of types. The library predicate `All` does exactly that: it “maps” a `Set`-typed function over a list, which yields (the type of) a heterogeneously typed list.

For example, the value interpretation of a type stack is a value stack where corresponding elements t and v are related by the type interpretation, that is, $v : \llbracket t \rrbracket$.

$$\begin{array}{ll} \text{Int} : \text{Stack} \rightarrow \text{Set} & \text{a-stack} : \text{Int} (\text{nat} :: \text{unit} :: \text{option addr} :: []) \\ \text{Int} = \text{All } \llbracket _ \rrbracket & \text{a-stack} = 42 :: \text{tt} :: \text{nothing} :: [] \end{array}$$

The definition of a program state (see Figure 3) abstracts over a `Mode` which contains a type interpretation that allows us reuse the same structure for concrete execution and abstract execution. A program state contains the program that is currently executed, the stack, and an environment which provides the context information to execute blockchain instructions like `AMOUNT` and `BALANCE`. It is parameterized by the output stack type, which does not change during execution. When executing more than one contract as we demonstrate in Sec. 3.4, this parameterization ensures that the results from completed contract executions are well typed.

The function `prog-step` executes the first instruction of a program on the current state. We explain two exemplary cases shown in Figure 3. To explain the first stanza of the code we have to make a confession. As several instructions have very similar semantics, our internal representation of instructions is a refinement of the datatype shown in Figure 2. For example, all instructions that just apply a function to the top of the stack are grouped under a constructor `fct` and `func-type` is the type defining these instructions.

$$\text{fct} : \text{func-type} \text{ args results} \rightarrow \text{Instruction}^+ \text{ args } [\times \text{ results}]$$

The function `app-fct` applies such a function to a concrete stack. Roughly speaking, if the underlying function has type $a_1 \rightarrow \dots \rightarrow a_n \rightarrow (r_1 \times \dots \times r_m)$ it gets transformed into a function between heterogeneously typed lists $[a_1, \dots, a_n] \rightarrow [r_1, \dots, r_m]$. We elide the

definition and just remark that the function $[\times _]$ implements the transformation between $(r_1 \times \dots \times r_m)$ and $[r_1, \dots, r_m]$. The functions `H.front` and `H.rest` (in Fig. 3) split the input stack according to the stack types expected by the function `ft`. The function `H.++` is concatenation of heterogeneous lists.

The `DROP` instruction drops the top of the stack.

3.2 Execution of Control Flow Instructions

We have chosen a small-step semantics because its stepwise progression matches the stepwise proof rules of the dynamic logic. However, the Michelson specification defines the semantics in terms of a big-step judgment.⁵

```
record Configuration (ri : Stack) : Set where
  constructor Conf
  field cenv : CEnvironment ; stk : Int ri

  data [_._]↓ : Configuration ri → Program ri ro → Int ro → Set
```

It relates a configuration (environment and input stack of type ri) and a program to an output stack of type ro . The definition of the semantics in the Michelson specification takes some liberties that require some extra machinery in a small-step execution model. We discuss these issues with some representative instructions.

The instruction `IF-NONE p-none p-some` expects a value of type `option` on top of the stack. If that value is `nothing` (the encoding of `NONE`), the $p\text{-}none$ branch is executed on the rest of the stack:

```
↓-IF-NONE : ∀ {p-none : Program txs tys} {p-some : Program (tx :: txs) tys}
  → [ Conf ce xs , p-none ]↓ ys
  -----
  → [ Conf ce (nothing :: xs) , IF-NONE p-none p-some ]↓ ys
```

If however the top of the stack is `just x` (encoding `SOME x`), the $p\text{-}some$ branch is executed on the stack where `just x` is replaced with x :

```
↓-IF-SOME : ∀ {p-none : Program txs tys} {p-some : Program (tx :: txs) tys}
  → [ Conf ce (x :: xs) , p-some ]↓ ys
  -----
  → [ Conf ce (just x :: xs) , IF-NONE p-none p-some ]↓ ys
```

To specify the corresponding small-step rule we introduce a type-respecting concatenation operator $\text{;}\bullet$ on programs. The program `IF-NONE p-none p-some ; p-rest` either transitions to $p\text{-}none ;\bullet p\text{-}rest$ or to $p\text{-}some ;\bullet p\text{-}rest$, depending on the value on top of the stack.

The instruction `DIP n p` executes program p on the stack that results from removing the first n elements of the current stack and reattaches them afterwards.

```
↓-DIP : ∀ {n} {q : T (n ≤b length txs)} {p-dip : Program (drop n txs) tys}
  → [ Conf ce (H.drop n xs) , p-dip ]↓ ys
  -----
  → [ Conf ce xs , DIP n {q} p-dip ]↓ (H.take n xs H.++ ys)
```

⁵ For typing reasons the implementation splits it in four judgments for programs \Downarrow , instructions \downarrow , shadow programs \Downarrow , and shadow instructions \downarrow^t .

3:10 Dynamic Logic for Symbolic Execution for Michelson

In the small-step version, dropping the first n elements of the stack is easy, but reattaching them requires extra machinery. Thus, a mechanism for holding on to the top of the stack while executing the subprogram and retrieving it afterwards is necessary.

Execution of **ITER** requires the same feature in a slightly different way. It consumes the list on top of the current stack. If the list is empty, it is dropped from the stack:

$$\begin{aligned} \downarrow\text{-ITER-NIL} : & \forall \{p\text{-iter} : \text{Program } (t :: t_{xs})\ t_{xs}\} \\ & \dashline \\ & \rightarrow [\text{Conf } ce ([\] :: xs), \text{ITER } p\text{-iter}] \downarrow xs \end{aligned}$$

Otherwise the subprogram is applied to the first list element v and then the **ITER** instruction is reissued on the rest of the list vs and the current stack:

$$\begin{aligned} \downarrow\text{-ITER-CONS} : & \forall \{v : [\] t\} \{vs : [\] \text{list } t\} \{xs\ ys\ zs : \text{Int } t_{xs}\} \{p\text{-iter} : \text{Program } (t :: t_{xs})\ t_{xs}\} \\ & \rightarrow [\text{Conf } ce (v :: xs), p\text{-iter}] \Downarrow ys \\ & \rightarrow [\text{Conf } ce (vs :: ys), \text{ITER } p\text{-iter}] \Downarrow zs \\ & \dashline \\ & \rightarrow [\text{Conf } ce ((v :: vs) :: xs), \text{ITER } p\text{-iter}] \Downarrow zs \end{aligned}$$

The typing for **ITER** requires that the type of the underlying stack is preserved, but the subprogram $p\text{-iter}$ is entitled to access and modify the stack beyond the first element x . Let's now consider stepwise execution. If the list on top has the form $v :: vs$, we need to stash the tail list vs somewhere while the subprogram processes the stack with v on top. After execution of the subprogram, we have to recover vs and try again with **ITER**.

As subprograms can be arbitrarily complex, in particular, they may contain **DIP** and **ITER**, we need a nestable solution. To this end, we add a single new instruction **MPUSH1** that pushes a single value on the stack. This instruction is different from the normal **PUSH** instruction, which is limited to **Pushable** values that have a textual representation.

$$\begin{aligned} \text{data } \text{ShadowInst } \{\mathcal{M} : \text{Type} \rightarrow \text{Set}\} : \text{Stack} \rightarrow \text{Stack} \rightarrow \text{Set} \text{ where} \\ \text{MPUSH1} : \forall \{t : \text{Type}\} \rightarrow \mathcal{M} t \rightarrow \text{ShadowInst } rS (t :: rS) \end{aligned}$$

We call the new instruction a *shadow instruction* because it does not appear in input programs. It is indexed by two stack types like any other instruction. A *shadow program* is defined like **Program**, but its first instruction can be a normal instruction or a shadow instruction. Shadow programs only appear at the top-level, never as subprograms nested in instructions. We elide the definition of **ShadowProg** as it is analogous to **Program**. Moreover, we provide a utility function **mpush** to generate a sequence of **MUSH1** instructions from a list of values.

$$\begin{aligned} \text{mpush} : & \forall \{\mathcal{M} : \text{Type} \rightarrow \text{Set}\} \{ri\} \{ro\} \{front : \text{Stack}\} \\ & \rightarrow \text{All } \mathcal{M} \ front \rightarrow \text{ShadowProg}\{\mathcal{M}\} (\text{front} ++ ri) ro \rightarrow \text{ShadowProg}\{\mathcal{M}\} ri ro \\ \text{mpush } [\] & sp = sp \\ \text{mpush } (x :: xs) & sp = \text{mpush } xs (\text{MPUSH1 } x \bullet sp) \end{aligned}$$

The small-step version of **DIP** n dp takes the top n elements from the stack and starts executing the program dp followed by the new instruction **mpush** $front$ where $front$ is the list of the n values that were removed from the stack.

$$\begin{aligned} \text{prog-step } \rho \mid \text{DIP } n\ dp ; p \\ = \text{record } \rho \{ \text{prg} = dp ; \bullet \text{mpush } (\text{H.take } n (\text{stk } \rho)) p ; \text{stk} = \text{H.drop } n (\text{stk } \rho) \} \end{aligned}$$

```
example-ITER : Program [ list nat ; nat ] [ nat ]
example-ITER = ITER (ADDnn ; end) ; end
```

■ **Figure 4** Simple program using **ITER**.

■ **Table 1** Program states during execution of Figure 4.

rSI	prg
[18 , 24] :: 0 :: []	ITER (ADD)
18 :: 0 :: []	ADD ; MPUSH [24] ; ITER (ADD)
18 :: []	MPUSH [24] ; ITER (ADD)
[24] :: 18 :: []	ITER (ADD)
24 :: 18 :: []	ADD ; MPUSH [] ; ITER (ADD)
42 :: []	MPUSH [] ; ITER (ADD)
[] :: 42 :: []	ITER (ADD)
42 :: []	end

The small-step version of **ITER** *ip* just pops the stack if the list is empty. Otherwise, if the top contains *v :: vs*, it pops this value, puts *v* on top of the stack and executes *ip* followed by **mpush** [*vs*] and then **ITER** *ip* and the rest of the program.

```
prog-step  $\rho$  | ITER ip ; p with stk  $\rho$ 
... | [] :: rsi = record  $\rho$  { prg = p ; stk = rsi }
... | (v :: vs) :: rsi = record  $\rho$  { prg = ip ;• (MPUSH1 vs • (ITER ip ; p)) ; stk = v :: rsi }
```

For illustration, Table 1 gives the stacks and shadow program of each intermediate state resulting from applying **prog-step** to the program in Figure 4 until program termination for the given input stack interpretation (omitting **end** for readability). This program adds a list of numbers on top of the stack to the number below.

3.3 Relation to Big-Step Semantics

Executing a program requires iterating the **prog-step** function. Our implementation drives this iteration by a step counter that is counted down at each instruction.

```
prog-step* :  $\mathbb{N} \rightarrow \text{CProgState } ro \rightarrow \text{CProgState } ro$ 
prog-step* zero  $\rho = \rho$ 
prog-step* (suc n)  $\rho = \text{prog-step* } n \text{ (prog-step } \rho\text{)}$ 
```

We prove that the original big-step semantics and our small-step semantics are equivalent in the usual sense.

```
bigstep $\Rightarrow$ smallstep :  $\forall (prg : \text{ShadowProg } txs tys)$ 
   $\rightarrow$  [ Conf ce xs , prg ] $\Downarrow$  ys
   $\rightarrow$   $\exists [ n ] \text{ prog-step* } n \text{ (cstate ce prg } xs\text{)} \equiv \text{cstate ce end } ys$ 

smallstep $\Rightarrow$ bigstep :  $\forall n \rightarrow (prg : \text{ShadowProg } txs tys) \rightarrow \{xs : \text{Int } txs\} \{ys : \text{Int } tys\}$ 
   $\rightarrow$  prog-step* n (cstate ce prg xs)  $\equiv$  cstate ce end ys
   $\rightarrow$  [ Conf ce xs , prg ] $\Downarrow$  ys
```

3:12 Dynamic Logic for Symbolic Execution for Michelson

```

record PrgRunning (Mode : MODE) : Set where
  constructor pr
  field {pp ss x y} : Type
    self      : Contract Mode pp ss
    sender    : Contract Mode x y
    p         : ProgState Mode [ pair (list operation) ss ]

record Transaction (Mode : MODE) : Set where
  constructor _,_
  field pops   : ( $\mathcal{M}$  Mode) (list operation)
  psender : Addr

data RunMode (Mode : MODE) : Set where
  Run : PrgRunning Mode → RunMode Mode
  Cont :  $\mathcal{F}$  Mode → RunMode Mode
  Fail :  $\mathcal{G}$  Mode → RunMode Mode

record ExecState (Mode : MODE) : Set where
  constructor exc
  field accounts : Blockchain Mode
  MPstate : RunMode Mode
  pending : List (Transaction Mode)

```

■ **Figure 5** Contract execution state.

3.4 Contract Execution and Execution Chains

The `prog-step` function can execute any Michelson program, not only those that comply to the typing restrictions of a contract. But it does not provide a mechanism to update the blockchain after successful contract execution nor one to execute other blockchain operations which might be emitted by a contract.

To implement these aspects of contract execution, the `ProgState` is augmented with further information as shown in Figure 5. The record `PrgRunning` holds the contracts involved in the current execution: `self` is the current contract and `sender` is the sender (the account that started the current contract). The `ExecState` holds the `Blockchain`, where contract execution results are saved, and a list of pending blockchain transactions to be executed. A value of type `Transaction` comprises a list of operations and the address of the sender of these operations. The field `MPstate` encodes the current mode of execution. `Run` indicates that a contract is currently executing the program in `PrgRunning` where we can take a step. `Cont` indicates the transition between one contract and the next; execution proceeds with the next pending blockchain operation. The \mathcal{F} argument is used by the abstract execution to propagate information between contract invocations. `Fail` indicates a failure along with an error code in its \mathcal{G} argument.

```

exec-step σ@(exc accts (Run (pr self _ (state en end [ new-ops , new-storage ] _))) pend)
= record σ{ accounts = set (self-address en) (upd-storage self new-storage) accts
; MPstate = Cont tt
; pending = (new-ops , self-address en) :: pend }
exec-step σ@(exc _ (Run pr@(pr _ _ ρ)) _)
= record σ{ MPstate = Run (record pr{ ρ = prog-step ρ }) }

```

Figure 6 Program execution.

The function `exec-step : CExecState → CExecState` maps an execution state to its successor state just like `prog-step` did for program states. It only implements the features mentioned above that cannot be modeled by the program state alone. Its definition is too big to include it in full; instead we briefly explain its implementation, giving each case in the same order as in the implementation.

Figure 6 contains the cases when a contract is executing.

1. When execution of the current contract has terminated (i.e., `MPstate` is `Run pr` and `ProgState.prg` matches `end`), then intrinsic typing ensures that the stack interpretation contains the emitted blockchain operations `new-ops` paired with the new storage value `new-storage`. We add the emitted operations to the `pending` field, update the terminated contract’s storage on the blockchain, and switch to `RunMode Cont`.
2. In all other cases of a running program, its `ProgState` evolves using `prog-step`.

In the remaining cases `MPstate` is `Cont tt` which means that no contract is currently executed. In this case `pending` is checked for other operations to be executed. Our model only implements the `TRANSFER-TOKENS` operation that initiates a new contract execution. We perform the following checks in this case:

- we fail unless the operation was emitted from a valid account;
- we fail unless the type of the parameter matches the input type of the called contract;
- we fail unless the target is a valid account;
- we fail unless the sender’s balance contains sufficient tokens to support the transfer.

The first three cases can never occur during an actual execution of a Michelson smart contract execution chain: The `TRANSFER-TOKENS` instruction only works for values of type `contract t`, which ensures validity of the target address and that the parameter type is `t`. Moreover, operations can only be emitted by valid accounts. The checks are needed in our model because it does not maintain information about which addresses are valid contract addresses. We chose not to include this information as it adds complexity without contributing to our goal of proving the soundness of symbolic execution.

4 Dynamic Logic for Michelson

To obtain a dynamic logic suitable for symbolic execution we follow the Key approach [9] and extend first order logic with a modality $[p]$, where p is a program state. The intuitive meaning is that $[p]\Psi$ holds for a formula Ψ , if running p terminates in a state such that Ψ holds. That is, the formula $\Phi \rightarrow [p]\Psi$ has a similar meaning as the Hoare triple $\{\Phi\} p \{\Psi\}$.

In the following, we concentrate on the proof rules for the modality. For instance (and ignoring the details of the program state for now), $\Phi \rightarrow [end]\Psi \equiv \Phi \rightarrow \Psi$ if the program is empty. Many simple proof rules have the form $\Phi \rightarrow [i; p]\Psi \equiv \Phi_i \wedge \Phi \rightarrow [p]\Psi$ where the formula Φ_i describes the effect of instruction i . If the instruction is a branch instruction on a predicate Q , like `if Q p1 p2`, the resulting formula is a disjunction as in $\Phi \rightarrow [(if Q p1 p2); p]\Psi \equiv Q \wedge \Phi \rightarrow [p1; p]\Psi \vee \neg Q \wedge \Phi \rightarrow [p2; p]\Psi$.

```

data _ $\vdash$ _ ( $\Gamma$  : Context) : Type → Set where
  var   :  $t \in \Gamma \rightarrow \Gamma \vdash t$ 
  const : [[ base bt ]] →  $\Gamma \vdash \text{base } bt$ 
  contr :  $\forall \{P : \text{Passable } t\} \rightarrow \text{Addr} \rightarrow \Gamma \vdash \text{contract } P$ 
  func  : 1-func args result → Match  $\Gamma$  args →  $\Gamma \vdash result$ 

data Formula ( $\Gamma$  : Context) : Set where
  'false  : Formula  $\Gamma$ 
  _:=_   :  $t \in \Gamma \rightarrow \Gamma \vdash t \rightarrow \text{Formula } \Gamma$ 
  _<_m_ : mutez ∈  $\Gamma \rightarrow \text{mutez } \in \Gamma \rightarrow \text{Formula } \Gamma$ 
  _≥_m_ : mutez ∈  $\Gamma \rightarrow \text{mutez } \in \Gamma \rightarrow \text{Formula } \Gamma$ 

```

■ **Figure 7** Terms and formulas.

We start by defining the formulas of the logic in Subsection 4.1.

4.1 Terms and Formulas

At the core of any symbolic execution there are symbolic (i.e., logical) variables representing the unknown operands. We represent such variables by a typed deBruijn index into a given Context = List Type. An abstract stack is then a list of typed variables:

```

Match : Context → Stack → Set
Match  $\Gamma$  = All (_ $\in$   $\Gamma$ )

```

Any knowledge that we have about the values on the stack is encoded in the list of formulas (over the variables on the stack) that we maintain in the program state. Figure 7 shows the terms and formulas used for the logic. Term comprise variables, constants of base type and of contract type, and simple functions. Here, “function” stands for proper functions as well as data constructors. For convenience, we restrict function arguments to variables and rely on variable equality in the formulas to specify complex terms.

As an example for the interplay between context, stack, and formulas, suppose the context defines three variables of type nat like this $\Gamma_1 = \text{nat} :: \text{nat} :: \text{nat} :: []$. An abstract stack for this context might just contain a single variable $x = 0$, where the 0 refers to the first variable in Γ_1 .

```

a-stack : Match  $\Gamma_1$  (nat :: [])
a-stack = [ x ]

```

If we further want to enforce that $x = y + 3$ (on natural numbers), then we have to encode that in two simple formulas, one that associates 3 to variable v , and another that states $x = y + v$. We do not impose a constraint on y , so it serves as an unconstrained symbolic variable.

```

x=y+3 : List (Formula  $\Gamma_1$ )
x=y+3 = x := func 'ADDnn (y :: v :: [])
          :: v := const 3
          :: []

```

Formulas are mainly used to express equality of a variable with a term. The inequalities express the ordering on tokens. The latter is used for token transfers where we have to know that the sender has sufficiently many tokens to satisfy the requirements of the transfer. The reader may wonder about conjunction and disjunction: the proof rules only generate them in the form of a disjunction of conjunctions of simple formulas. We represent this structure as a list of lists of simple formulas. Repetition does not matter in this representation for two reasons: 1. disjunction and conjunction are both idempotent; 2. we are only interested in validity of a formula, but do not transform it in any way.

4.2 Representing Michelson Program State in DL

We simplify the handling of formulas of the form $\Phi \rightarrow [p]\Psi$ by reusing our previous definition of the type `ProgState` in a different mode as an *abstract* state.

```
AMode : Context → MODE
AMode Γ = record { M = _ ∈ Γ
                    ; F = List (Formula Γ)
                    ; G = List (Formula Γ) ↗ List (Formula Γ)
                    }
```

That is, we replace the normal representation of values in \mathcal{M} by symbolic variables, in \mathcal{F} we maintain a list (i.e., conjunction) of formulas, and in \mathcal{G} we maintain a tagged list of formulas to represent different modes of failure.

The meaning of an abstract state is a conjunction that specifies the value for `AMOUNT` and `BALANCE` in the environment, it specifies the size of the stack and all values on it, and it collects further constraints generated by application of the proof rules.

Informally, an abstract program state represents $\Theta \implies [prg]\Psi$ where

$$\Theta \equiv \text{state of environment} = \text{en} \wedge \text{state of stack} = \text{stk} \wedge \bigwedge_{\phi \in \Phi} \phi$$

The encoding of the implication in the abstract program state corresponds exactly to the abstract instance of the `ProgState` type (see Figure 3). Reusing the type in this way makes the formalization of symbolic execution very similar to the concrete execution model presented in Section 3. This similarity in turn makes the soundness proof easier and more concise. All constructs for concrete execution are reused in the abstract by instantiating their `MODE` parameter. Thus, they are automatically parameterized by a `Context` Γ and the names of the structures are the same as for concrete execution but prefixed with an α (only the abstract blockchain is called β lockchain).

Symbolic execution of control flow can lead to disjunctions over such states, which is represented using a list of abstract program states. Each of the branch comes with its own state, which requires existential quantification over the types of the variables in Γ .

```
⊕Prog-state : Stack → Set
⊕Prog-state ro = List (Ξ[ Γ ] αProg-state Γ ro)
```

Using Agda lists to represent conjunctions and disjunctions is convenient for two reasons.

1. Conjunctions and disjunctions do not mix: Φ always represents a conjunction over its elements and disjunctions can only occur as a result of some symbolic execution rules that implement control flow. In this case, the disjunction always affects every aspect of the abstract program state (i.e., the remaining programs will always differ).
2. Agda’s “element of” relation for lists makes the implementation of the rules of the calculus simple and efficient.

4.3 Proof Rules for Michelson

The rules for symbolic execution are formalized by a function that maps an abstract program state into a set (list) of abstract program states.

$$\alpha_{\text{prog-step}} : \forall \{\Gamma ro\} \rightarrow \alpha_{\text{Prog-state}} \Gamma ro \rightarrow \cup_{\text{Prog-state}} ro$$

It mimics `prog-step` and gives a deterministic way of symbolic execution. Every (non-environmental) functional instruction can be executed concretely with a single rule as shown in Figure 6. During symbolic execution, the only thing that is guaranteed is that the stacks contain values of the expected type. For example, if the next instruction is `ADDnn`, we can conclude that there are two values of type `nat` on top of the stack before the instruction and one value of type `nat` afterwards. Moreover, we can say that this value is the sum of the two values that were on top of the stack before, but we have to express that with a constraint, i.e., a logical formula.

That is, symbolic execution of `ADDnn` introduces a new variable v_r that replaces the variables v_x and v_y from the top of the stack, and adds a clause that equates this new variable with the sum of the former two:

$$v_r := \text{ADDnn } v_x v_y$$

In this way, we can give a single symbolic execution rule for all functional instructions that return a single result.

$$\begin{aligned} \alpha_{\text{prog-step}} \{\Gamma\} (\text{state } \alpha_{\text{en}} (\text{fct } (\text{D1 } \{result = result\} f) ; prg) \alpha_{\text{st}} \Phi) \\ = [(result :: \Gamma) \\ , \text{state } (\text{wk}\alpha_{\text{E}} \alpha_{\text{en}}) (\text{wkSP } prg) (0\in :: \text{wkM } (\text{H.rest } \alpha_{\text{st}})) \\ (0\in := \text{wkL- } (\text{func } f (\text{H.front } \alpha_{\text{st}})) :: \text{wk}\Phi \Phi)] \end{aligned}$$

Let's decompress this definition. We pattern match against the current (abstract) state to obtain the environment α_{en} , the current instruction, the rest of the program prg , the stack α_{st} , and the formula Φ . The constructor `fct` indicates a functional instruction and the constructor `D1` indicates that f returns a single result of type `result`.

As the instruction does not implement any control flow, there is only a single next state. Its description starts with the extended context $result :: \Gamma$, which introduces a new variable of type `result` for the result. The name, rather the deBruijn address, of this variable is `0E`, which denotes the first entry in the context. The second component describes the new state, which (ignoring the `wk` functions for the moment) keeps the environment, moves to the rest of the program, pushes the result on the stack after removing the arguments using `H.rest`, and pushes a new equation that defines the value of `0E` as the result of applying f to the front of the stack. The functions `H.front` and `H.rest` operate on heterogenous lists and are defined such that $\alpha_{\text{st}} \equiv \text{H.front } \alpha_{\text{st}} \text{ H.++ H.rest } \alpha_{\text{st}}$ where the actual division is driven by the type of f . The operation `H.++` is concatenation of heterogenous lists. The `wk` functions are a consequence of using deBruijn indices for variables: if we introduce new variables, all existing variables have to be incremented by the number of new variables (i.e., weakened). We do not show their definition, as this manipulation of deBruijn indices is standard.

We do not have a general mechanism for the other functional instructions (see Figure 8), as they behave very differently in a symbolic context: `UNPAIR` requires two new variables and clauses, while `SWAP` only changes the position of two stack values. No new variables or clauses are necessary because `SWAP` only reconfigures the stack.

The instruction `PUSH` needs special treatment because it can handle arbitrarily complex compound values. When pushing a value x of primitive type, it is sufficient to add a new variable and a clause which sets this variable equal to the term `const x`. But if x has a list

```

 $\alpha\text{prog-step } \{\Gamma\} (\text{state } \alpha en (\text{fct } (\text{Dm } ('UNPAIR \{t1\} \{t2\})) ; prg) (p \in :: \alpha st) \Phi)$ 
 $= [ (t1 :: t2 :: \Gamma)$ 
 $, \text{state } (\text{wk}\alpha E \alpha en) (\text{wkSP } prg) (0 \in :: 1 \in :: \text{wkM } \alpha st)$ 
 $(0 := \text{func } 'CAR [ \text{wkE } p \in ] :: 1 := \text{func } 'CDR [ \text{wkE } p \in ] :: \text{wk}\Phi \Phi )]$ 

 $\alpha\text{prog-step } \alpha @ (\text{state } \alpha en (\text{fct } (\text{Dm } 'SWAP) ; prg) (x \in :: y \in :: \alpha st) \Phi)$ 
 $= [ -, \text{record } \alpha \{ \text{prg} = prg ; \text{stk} = y \in :: x \in :: \alpha st \} ]$ 

 $\alpha\text{prog-step } \{\Gamma\} (\text{state } \alpha en (\text{fct } ('PUSH } P x) ; prg) \alpha st \Phi)$ 
 $= [ (\text{expand}\Gamma P x ++ \Gamma)$ 
 $, \text{state } (\text{wk}\alpha E \alpha en) (\text{wkSP } prg) ((\text{ewk } (0 \in \text{ex}\Gamma P)) :: \text{wkM } \alpha st)$ 
 $(\Phi \text{wk } (\text{unfold } P x) ++ \text{wk}\Phi \Phi)]$ 

```

■ **Figure 8** Functional instructions (excerpt).

type or an option type, its value cannot be expressed with a `const` term. In general, the symbolic execution of a single `PUSH` instruction may create arbitrarily many (but linear in the size of the pushed value) new variables and clauses.

To this end, the function `unfold` $P x$ creates all clauses required to express the value x . This process defines a list of new variables of types defined by `expand Γ` $P x$.⁶ For example, `PUSH {list ty} P (y :: ys)` gives rise to two new variables r_y of type `ty` for y and r_{ys} of type `list ty` for ys and an equation $r := \text{func } (\text{CONS } [r_y, r_{ys}])$, where r is the variable for the result. The function `unfold` proceeds recursively: if $ys = []$, its variable can be set to `func (NIL ty) []`, otherwise it will be further decomposed. Similarly for y : if ty is a primitive type, it can be set to `const y`, otherwise it must be further decomposed as well.

As an example, we show the result of unfolding the list $[0, 1] : \text{list nat}$. The generated context is $\Gamma_2 = \text{list nat} :: \text{list nat} :: \text{list nat} :: \text{nat} :: \text{nat} :: []$ and the generated list of equations to represent the list is as follows.

```

eqn : List (Formula  $\Gamma_2$ )
eqn = c1 := func 'CONS (x0 :: c2 :: [])
:: c2 := func 'CONS (x1 :: c3 :: [])
:: c3 := func ('NIL nat) []
:: x0 := const 0
:: x1 := const 1
:: []

```

We finish with the abstract execution of the conditional instruction `IF-NONE` (see Figure 9). This instruction expects a value of type `option t` on top of the stack. Here we have two possible next states, depending on whether the value is present. The first disjunct deals with the case where the value is `NONE`. In this case, the stack is popped, the `thn` branch is taken, and the equation enforcing the value to be `NONE` is added. There are no new variables, so there is no weakening in this disjunct.

⁶ We do not include the tedious definitions of these auxiliary functions here, but encourage the interested reader to check the supplementary material.

```


$$\alpha_{\text{prog-step}} \{\Gamma\} (\text{state } \alpha_{en} (\text{IF-NONE } \{t = t\} \text{ thn } \text{els} ; \text{prg}) (\text{o} \in :: \alpha_{st}) \Phi) \\
= [\Gamma, \text{state } \alpha_{en} (\text{thn} ; \bullet \text{prg}) \alpha_{st} (\text{o} \in := \text{func} (\text{'NONE } t) [] :: \Phi) \\
; (t :: \Gamma), \text{state } (\text{wk}\alpha_{\text{E}} \alpha_{en}) (\text{els} ; \bullet \text{wkSP prg}) (\text{o} \in :: \text{wkM } \alpha_{st}) \\
(\text{wk}\in \text{o} \in := \text{func} \text{'SOME } [0 \in] :: \text{wk}\Phi \Phi)]$$


```

Figure 9 Symbolic execution of IF-NONE.

The second disjunct models the case where the value on top of the stack is **SOME** y . Here we need a new variable of type t for y , pop the stack and push the new variable, we take the els branch, and add an equation that forces the value to be **SOME** y .

4.4 Proof Rules for the Blockchain Run-time

Just like the symbolic execution rules for the Michelson DL, those for the DL on blockchain operations are given analogously.

```

\alpha_{\text{ExecState}} : \text{Set} \\
\alpha_{\text{ExecState}} = \text{List } (\exists[\Gamma] \alpha_{\text{ExecState}} \Gamma)

```

```

\alpha_{\text{exec-step}} : \forall \{\Gamma\} \rightarrow \alpha_{\text{ExecState}} \Gamma \rightarrow \alpha_{\text{ExecState}}

```

The switch from concrete to abstract execution state is achieve by changing the *Mode* parameter of the **ExecState** (see Figure 5). Its \mathcal{F} field replaces concrete semantics by abstract semantics throughout all state components.

Unfortunately $\alpha_{\text{exec-step}}$ cannot represent **exec-step** exactly, if **MPstate** is **Cont** Φ , that is: a contract has terminated and we need to check the **pending** field for further operations to be executed. At this point, the predicate Φ has to supply sufficient information about the values of the variables representing the pending operations to proceed in a meaningful way. The **pending** list contains pairs of a list of operations and a sender address. While the latter is a concrete address, the former is a variable of type **list operation** $\in \Gamma$. To proceed, we have to know if the list is empty (so that we can proceed to the next block of pending operations) or not. In the latter case, we need to ensure that the first element of the operation list is a **TRANSFER-TOKENS**, and so on.

To this end, we defined several auxiliary functions to inspect the constraints in Φ for patterns that restrict the models sufficiently. For example, the function **find-tt-list** takes a conjunction of formulas and a variable of type **list t** and tries to find a formula that restricts this variable to **NIL** or **CONS**:

```

\text{find-tt-list} : \forall \{\Gamma\}\{t\} \rightarrow \text{List } (\text{Formula } \Gamma) \rightarrow \text{list } t \in \Gamma \\
\rightarrow \text{Maybe } (\text{Match } \Gamma [] \uplus \text{Match } \Gamma [t ; \text{list } t])

```

```

\text{find-tt-list-soundness} : \forall \{\Gamma\}\{t\} \rightarrow (\Phi : \text{List } (\text{Formula } \Gamma)) \rightarrow (l \in : \text{list } t \in \Gamma) \\
\rightarrow \text{find-tt-list } \Phi l \in \equiv \text{just } (\text{inj}_1 [])
\rightarrow \forall (\gamma : \text{Int } \Gamma) \rightarrow \gamma \models \Phi \\
\rightarrow \text{lookup } \gamma l \in \equiv []

```

$$\begin{aligned}
 \text{val}\vdash : \forall \{ty \Gamma\} \rightarrow \text{Int } \Gamma \rightarrow \Gamma \vdash ty \rightarrow \llbracket ty \rrbracket \\
 \text{val}\vdash \gamma (\text{var } v \in) &= \text{lookup } \gamma v \in \\
 \text{val}\vdash \gamma (\text{const } b) &= b \\
 \text{val}\vdash \gamma (\text{contr } adr) &= adr \\
 \text{val}\vdash \gamma (\text{func } f \text{ args}) &= \text{appD1 } f (\text{map } (\text{lookup } \gamma) \text{ args})
 \end{aligned}$$

$$\begin{aligned}
 _ \models \varphi : \forall \{\Gamma\} \rightarrow \text{Int } \Gamma \rightarrow \text{Formula } \Gamma \rightarrow \text{Set} \\
 \gamma \models \varphi \text{ 'false} &= \perp \\
 \gamma \models \varphi (v \in := trm) &= \text{lookup } \gamma v \in \equiv \text{val}\vdash \gamma trm \\
 \gamma \models \varphi (x <_m x_1) &= \text{lookup } \gamma x < \text{lookup } \gamma x_1 \\
 \gamma \models \varphi (x \geq_m x_1) &= \text{lookup } \gamma x \geq \text{lookup } \gamma x_1
 \end{aligned}$$

■ **Figure 10** Semantics of terms and formulas.

We only show the soundness lemma for **NIL**, as the one for **CONS** is analogous. This approach is not complete as the implementation of **find-tt-list** is tailored to the constraints as they are produced by symbolic execution.

The full implementation is quite involved and relies on several further lemmas that examine constraints (for example if the current balance of a sender is sufficient for a token transfer) in a similar way. We refer the interested reader to the supplement.

The remaining cases deal with a terminated contract execution where the new state is written back to the blockchain or the execution of an abstract program step for the contract under execution. The first case is similar to the concrete implementation where new variables are introduced for the updated values. The second case is more complicated because the context extensions from the abstract program step are encoded in the list of resulting disjunctions, so an additional term has to be supplied proving that these contexts are actually an extension of the original context.

5 Semantics and Soundness

5.1 Values and Models

As a context is just a list of types like a stack, its interpretation is also a heterogeneous list of values as defined by **Int**. For a given context interpretation γ , the semantics of a term and a formula is defined as usual (see Figure 10).

For a given context interpretation γ and abstract and concrete (program or execution) states, the predicates **modp** and **modσ** express that under this interpretation the given abstract state models the concrete state. This is the case when the formulas in Φ are true under γ and the real and variable values are the same for the stacks and every other element.

$$\begin{aligned}
 \text{MODELING} : \text{Context} \rightarrow (\text{MODE} \rightarrow \text{Set}) \rightarrow \text{Set}_1 \\
 \text{MODELING } \Gamma F = \text{Abstract } F \Gamma \rightarrow \text{Concrete } F \rightarrow \text{Set} \\
 \\
 \text{modp} : \forall \{\Gamma\} \rightarrow \text{Int } \Gamma \rightarrow \text{MODELING } \Gamma \lambda M \rightarrow \text{Prog-state } M \text{ ro} \\
 \text{modp } \gamma (\text{state } \{ri = \alpha r_i\} \alpha en \alpha prg rVM \Phi) \\
 \quad (\text{state } \{ri\} en prg stk tt) \\
 = \Sigma (\alpha r_i \equiv ri) \lambda \{ \text{refl} \rightarrow \\
 \quad \text{modE } \gamma \alpha en en \times \text{modprg } \gamma \alpha prg prg \times \text{modS } \gamma rVM stk \times \text{modF } \gamma \Phi \}
 \end{aligned}$$

```

soundness  $\gamma$  (state  $\alpha en$  (IF-NONE  $thn$   $els$  ;  $aprg$ ) ( $o \in :: rVM$ )  $\Phi$ )
           (state  $en$  (.IF-NONE  $thn$   $els$  ;  $cprg$ ) (just  $x :: stk$ ) tt)
           (mod $\rho$ (  $mE$  , ( $o \equiv$  ,  $mrS$ ) , (refl , refl , mPRG) ,  $m\Phi$  ))
=  $_$  , [  $x$  ] ,  $_$  ,  $1 \in$  , (refl , wkmodE  $mE$  , modprg-extend  $els$  (wkmodprg  $mPRG$ ) ,
           (refl , wkmodS  $mrS$ ) , ( $o \equiv$  , wkmod $\Phi$   $m\Phi$ ))

soundness  $\gamma$  (state  $\alpha en$  (IF-NONE  $thn$   $els$  ;  $aprg$ ) ( $o \in :: rVM$ )  $\Phi$ )
           (state  $en$  (.IF-NONE  $thn$   $els$  ;  $cprg$ ) (nothing ::  $stk$ ) tt)
           (mod $\rho$ (  $mE$  , ( $o \equiv$  ,  $mrS$ ) , (refl , refl , mPRG) ,  $m\Phi$  ))
=  $_$  , [] ,  $_$  ,  $0 \in$  , (refl ,  $mE$  , modprg-extend  $thn$   $mPRG$  ,  $mrS$  , ( $o \equiv$  ,  $m\Phi$ ))

```

■ **Figure 11** Prog-step soundness for IF-NONE (excerpt).

They all have a similar structure expressed by the **MODELING** function as they relate an abstract thing with a concrete thing. They are implemented by several auxiliary **modX** predicates for every subcomponent of program and execution states. For example, **ModE** relates execution environments, **modprg** relates shadow programs, **modS** relates stacks, and **mod Φ** checks that the formulas are all true. The definition of **mod σ** is similar.

To show that a disjunction of abstract states models a concrete state, we show that one of the states in the disjunction models the state:

```

mod $\Psi\sigma$  :  $\forall \{\Gamma\} \rightarrow \text{Int } \Gamma \rightarrow \Psi\text{ExecState} \rightarrow \text{CExecState} \rightarrow \text{Set}$ 
mod $\Psi\sigma$   $\{\Gamma\} \gamma \Psi\sigma \sigma = \exists [\alpha\sigma] (\Gamma, \alpha\sigma) \in \Psi\sigma \times \text{mod}\sigma \gamma \alpha\sigma \sigma$ 

```

5.2 Soundness of the DL

We prove the soundness of the logic by showing that when an abstract state models a concrete one, the result of one-step symbolic execution models the result from concrete execution of the same step. Here are the types of the proof terms for program steps and execution steps.

```

soundness :  $\forall \{\Gamma ro\} \gamma \alpha\rho \rho \rightarrow \text{mod}\rho \{ro\} \{\Gamma\} \gamma \alpha\rho \rho$ 
            $\rightarrow \exists [\Gamma'] \exists [\gamma'] \text{mod}\Psi\rho \{\Gamma = \Gamma' ++ \Gamma\} (\gamma' \text{H}.\text{++} \gamma) (\alpha\text{prog-step} \alpha\rho) (\text{prog-step} \rho)$ 

soundness :  $\forall \{\Gamma\} (\gamma : \text{Int } \Gamma) \rightarrow \forall \alpha\sigma \sigma \rightarrow \text{mod}\sigma \gamma \alpha\sigma \sigma$ 
            $\rightarrow \exists [\Phi] \text{ExecState.MPstate} \alpha\sigma \equiv \text{APanic} \Phi$ 
            $\Psi \exists [\Gamma'] \exists [\gamma'] \text{mod}\Psi\sigma \{\Gamma' ++ \Gamma\} (\gamma' \text{H}.\text{++} \gamma) (\alpha\text{exec-step} \alpha\sigma) (\text{exec-step} \sigma)$ 

```

The first **soundness** statement addresses soundness of **αprog-step**. As the modeling relation is mostly composed of equalities, the proof gets accepted by Agda, once we supply sufficiently precise arguments to match the cases in the definition of **αprog-step**. We pattern match against **refl** and parts of the arguments, as well as we show that the weakened parts of the formula are still modeled with the extended context (if new variables were introduced in the case).

Figure 11 shows the case for the **IF-NONE** instruction. Without going into details, it is easy to spot the handling of the concrete and abstract stack and that the outcome of the test determines which of the possibilities of the abstract outcome is chosen (cf. $0 \in$ and $1 \in$).

The most complicated case of this proof establishes soundness for any scalar function (see Figure 12). It works by showing that applying the front of the previous stack interpretation to the given function yields the same result as applying the extended interpretation of the top of the previous stack matching to it.

```

soundness  $\gamma$  (state  $\alpha en$  (fct (D1 f) ; aprg) rVM  $\Phi$ )
  (state  $en$  (.fct (D1 f) ; cprg) stk tt)
  (modp( mE , mrS , (refl , refl , mPRG) , m $\Phi$  ))
with modS++ rVM stk mrS
... | mfront , mrest =
let result = appD1 f (H.front stk) in
_ , [ result ] , _ , 0 $\in$  , (refl ,
wkmodE mE , wkmodprg mPRG , (refl , wkmodS mrest) ,
(cong (appD1 f) (trans (sym (modIMI mfront)) (wkIMI { $\gamma$ ' = [ result ]})) , wkmod $\Phi$  m $\Phi$ ))

```

■ **Figure 12** Prog-step soundness for scalar functions (excerpt).

The second **soundness** statement establishes soundness for those cases of **α exec-step** where a contract execution is active. This part appears simple because it only covers two cases: Either we are in the middle of running a contract, in which case we reuse the soundness proof for program state execution, or the current contract execution has terminated and we have to prove that the blockchain and the pending list are updated correctly. The first case is straightforward, but tedious because we need to copy parts of the previous proof. The second case is fairly technical as it involves getting the proof in sync with the definitions of concrete and abstract execution.

6 Related Work

Research on formal verification of blockchain-based applications has experienced rapid growth in the last decade. Various techniques and frameworks have been applied to enhance the safety of smart contracts. In this section, we discuss some key approaches, particularly those employing symbolic execution in the context of smart contracts.

6.1 Verification of Smart Contracts

Symbolic execution is a powerful technique for systematically exploring program paths and identifying potential vulnerabilities in smart contracts. Most of the existing tools focus on the Ethereum platform. Tsankov et al. introduced SECURIFY [32], a tool that utilizes symbolic execution to perform practical security analysis on Ethereum smart contracts. It targets common vulnerability security patterns specified in a designated domain-specific language. SECURIFY symbolically encodes the dependence graph of the contract in stratified Datalog to extract semantic information from the code. After obtaining semantic facts, it checks whether the security patterns hold or not. Similarly, Manticore [26] and KEVM [18] use symbolic execution to analyze Ethereum smart contracts. KEVM is an executable formal specification built with the K Framework for the Ethereum virtual machine’s bytecode (EVM), a stack-based and low-level smart contract language for the Ethereum blockchain. Since tokens can hold a significant amount of value, they are often targeted for attacks. Therefore, several tools [18, 29] conduct case studies for the implementations of token standards.

Several approaches use existing formal verification frameworks to ensure the correctness and security of smart contracts. Amani et al. [3] proposed the formal verification of Ethereum smart contracts in Isabelle/HOL. Hirai [19] formalizes the EVM using Lem, a language to specify semantic definitions. The formal verification of smart contracts is achieved using

the Isabelle proof assistant. Mi-cho-Coq [10] is a framework for the proof assistant Coq to verify functional correctness of Michelson smart contracts. They formalize the semantics of a Michelson in Coq using a weakest precondition calculus and verify several contracts. It provides full coverage of the language whereas our goal is to give a blueprint for a soundness proof of symbolic execution.

There are several tools for automated verification including solc-verify [15], VerX [4], and Oyente [22]. solc-verify processes smart contracts written in Solidity and discharges verification conditions using modular program analysis and SMT solving. It operates at the level of the contract source code, with properties specified as contract invariants and function pre- and post-conditions provided as annotations in the code by the developer. This approach offers a scalable, automated, and user-friendly formal verification solution for Solidity smart contracts. The core of solc-verify involves translating Solidity contracts to Boogie IVL (Intermediate Verification Language), a language designed for verification.

Nishida et al. [27] developed HELMHOLTZ, an automated verification tool for Michelson. While both research efforts aim to build a verification tool for smart contracts written in Michelson, HELMHOLTZ is based on refinement types, whereas we consider symbolic execution. HELMHOLTZ has better coverage of Michelson instructions than we currently have, but it can only verify a single contract whereas our model and soundness proof covers full inter-contract verification. The HELMHOLTZ developers plan to extend Helmholtz with inter-contract behavior.

Bau et al. [8] implement a static analyzer for Michelson within the modular static analyser MOPSA that is based on abstract interpretation. It is able to infer invariants on a contract's storage over several calls and it can prove the absence of errors at run time.

Da Horta et al. [5] aim at automating as much of the verification process as possible by automatically translating a Michelson contract into an equivalent program for the deductive program verification platform WHY3. However, they found that sometimes user intervention was required, and their tool can only verify single contracts individually.

6.2 Symbolic Execution for Bytecode Interpretation

As there are some parallels between Michelson and bytecode languages, we discuss symbolic execution methods for some selected bytecode languages.

Albert et al. [2] transform Java bytecode into a logic program to utilize analysis techniques from logic programming, specifically symbolic semantics, for the formal verification of the bytecode. They verify properties such as termination and run-time error freeness and infer resource bounds. The dynamic aspects of bytecode, such as control flow and data flow, are effectively handled through the analysis performed on the logic program. Balasubramanian, Daniel et al. [7] include dynamic symbolic execution tailored for Java-based web server environments. Their tool analyzes the bytecode interactions within the Java Virtual Machine and focuses on bytecode instructions, method calls, object manipulations and memory interactions to detect vulnerabilities and bugs.

Several approaches address formal semantics and analysis for WebAssembly (Wasm) [24, 21, 34]. Watt, Conrad et al. [34] present Wasm Logic, a formal program logic for WebAssembly. The authors mechanize Wasm logic and its proof of correctness in Isabelle/HOL. To this end, they propose an alternative semantics. Just like our work (we propose a logic on an alternative semantics, mechanize it, and prove its soundness in Agda), their aim is to provide a logical basis for static analysis tools.

Marques et al. [24] present a concolic execution engine that systematically explores different program paths by combining concrete and symbolic execution to enable automated testing and fault detection. It models execution behavior and uses constraint solvers to

generate inputs and explore paths, taking into account the complexity of Wasm’s stack-based execution and binary format. Unlike our work, their work is geared towards implementing a realistic tool.

6.3 Related Uses of Dynamic Logic

The idea of using dynamic logic for symbolic execution can be traced back to Heisel et al. [17]. They formalize Burstall’s verification method [11] using symbolic execution and induction in the framework of dynamic logic.

Maingaud et al. [23] define a program logic for imperative ML programs based on dynamic logic and prove its soundness. Their goal is to use this logic as a basis for symbolic execution.

Similar to our approach, the research of Ahrendt et al. [1] emphasizes data integrity in Solidity smart contracts. This framework verifies smart contracts and ensures strong data integrity and functional correctness under various conditions. It introduces a specification language for defining contract properties and behaviors that are critical for security and reliability. Similar approaches to ours aim to verify the correctness and security of smart contracts, but differ in methodology and target languages. Their approach uses dynamic logic for invariant-based specifications with prototype-based tools, while our approach uses dynamic logic for symbolic execution and focuses on formal proofs.

Abstract execution [31] is a static verification framework based on symbolic execution. It is geared at schematic programs, i.e., programs with placeholders for program fragments, so that it can be used to prove certain program transformations correct. Its logical basis is dynamic logic extending earlier work for Java [9].

7 Conclusion

We presented a dynamic logic for Michelson as well as its extension to blockchain operations on a small but representative subset of Michelson. The goal was to create a core model that covers instances of all kinds of operations and that can be easily extended with further Michelson instructions. We achieved full coverage of scalar functional instructions, the majority of Michelson instructions. To include any further scalar instruction, one only has to specify its typing rule and its implementation in Agda. The symbolic execution rule and the soundness proof for that rule is already provided by our model. Further instructions that retrieve information from the execution environment can be added easily as well by extending the [Environment](#) record and its subcomponents to include such information.

We cover three exemplary instructions for control flow, because most other conditional and looping instructions are either very similar or very simple and thus easy to include in the presented model. One aspect of Michelson that is not covered is first-class functions. Including them might require some reworking of the current model to store such values on the stack.

Efficient symbolic execution is **not** a goal of this work: Agda can normalize a concrete or symbolic execution state to enable inspection of the state after one or more execution steps, but in our experiments normalization was sometimes infeasible after less than ten symbolic execution steps. Nevertheless, we plan to use our soundness proof as the basis for an efficient symbolic interpreter for Michelson in ongoing work.

References

- 1 Wolfgang Ahrendt and Richard Bubel. Functional verification of smart contracts via strong data integrity. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Applications*, pages 9–24, Cham, 2020. Springer International Publishing.
- 2 Elvira Albert, Miguel Gómez-Zamalloa, Laurent Hubert, and Germán Puebla. Verification of java bytecode using analysis and transformation of logic programs. In Michael Hanus, editor, *Practical Aspects of Declarative Languages*, pages 124–139, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
- 3 Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. Towards verifying Ethereum smart contract bytecode in Isabelle/HOL. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pages 66–77, January 2018. doi:10.1145/3167084.
- 4 Permenev Anton, Dimitrov Dimitar, Tsankov Petar, Dana Drachsler-Cohen, and Martin Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1661–1677, 2020. doi:10.1109/SP40000.2020.00024.
- 5 Luís Pedro Arrojado da Horta, João Santos Reis, Simão Melo de Sousa, and Mário Pereira. A tool for proving michelson smart contracts in why3. In *2020 IEEE International Conference on Blockchain (Blockchain)*, pages 409–414, 2020. doi:10.1109/Blockchain50366.2020.00059.
- 6 Barnabas Arvay, Thi Thu Ha Doan, and Peter Thiemann. Contract Orchestration for Michelson. Software, version 0.5 (visited on 2024-08-29). URL: <https://freidok.uni-freiburg.de/data/255176>.
- 7 Daniel Balasubramanian, Zhenkai Zhang, Dan McDermet, and Gabor Karsai. Dynamic symbolic execution for the analysis of web server applications in java. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC ’19, pages 2178–2185, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3297280.3297494.
- 8 Guillaume Bau, Antoine Miné, Vincent Botbol, and Mehdi Bouaziz. Abstract interpretation of michelson smart-contracts. In *Proceedings of the 11th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*, SOAP 2022, pages 36–43, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3520313.3534660.
- 9 Bernhard Beckert, Vladimir Klebanov, and Benjamin Weiß. Dynamic logic for java. In Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors, *Deductive Software Verification – The KeY Book: From Theory to Practice*, pages 49–106. Springer International Publishing, Cham, 2016. doi:10.1007/978-3-319-49812-6_3.
- 10 B. Bernardo, R. Cauderlier, Z. Hu, B. Pesin, and J. Tesson. Mi-Cho-Coq, a framework for certifying Tezos smart contracts. In *Formal Methods. FM 2019 International Workshops - Porto, Portugal, October 7-11, 2019, Revised Selected Papers, Part I*, volume 12232 of *Lecture Notes in Computer Science*, pages 368–379. Springer, 2019. doi:10.1007/978-3-030-54994-7_28.
- 11 Rod M. Burstall. Program proving as hand simulation with a little induction. In Jack L. Rosenfeld, editor, *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974*, pages 308–312. North-Holland, 1974.
- 12 Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008. URL: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf.
- 13 Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. Dysy: dynamic symbolic execution for invariant inference. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 281–290. ACM, 2008. doi:10.1145/1368088.1368127.

- 14 L. Goodman. Tezos-a self-amending crypto-ledger, 2014. URL: <https://www.tezos.com/static/papers/white-paper.pdf>.
- 15 Á. Hajdu and D. Jovanović. solc-verify: A modular verifier for solidity smart contracts. In S. Chakraborty and J. A. Navas, editors, *Verified Software. Theories, Tools, and Experiments*, pages 161–179. Springer International Publishing, 2020.
- 16 David Harel, Jerzy Tiuryn, and Dexter Kozen. *Dynamic Logic*. MIT Press, Cambridge, MA, USA, 2000.
- 17 Maritta Heisel, Wolfgang Reif, and Werner Stephan. Program verification by symbolic execution and induction. In Katharina Morik, editor, *GWAI-87, 11th German Workshop on Artificial Intelligence, Geseke, Germany, September 28 - October 2, 1987, Proceedings*, volume 152 of *Informatik-Fachberichte*, pages 201–210. Springer, 1987. doi:10.1007/978-3-642-73005-4_22.
- 18 Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. KEVM: A complete formal semantics of the Ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217, 2018. doi:10.1109/CSF.2018.00022.
- 19 Y. Hirai. Defining the Ethereum virtual machine for interactive theorem provers. In *Financial Cryptography and Data Security*, pages 520–535. Springer International Publishing, 2017.
- 20 James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976. doi:10.1145/360248.360252.
- 21 Daniel Lehmann and Michael Pradel. Wasabi: A framework for dynamically analyzing webassembly. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’19, pages 1045–1058, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3297858.3304068.
- 22 Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 254–269, 2016.
- 23 Séverine Maingaud, Vincent Balat, Richard Bubel, Reiner Hähnle, and Alexandre Miquel. Specifying imperative ML-like programs using dynamic logic. In Bernhard Beckert and Claude Marché, editors, *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2010, Paris, France, June 28-30, 2010, Revised Selected Papers*, volume 6528 of *Lecture Notes in Computer Science*, pages 122–137. Springer, 2010. doi:10.1007/978-3-642-18070-5_9.
- 24 Filipe Marques, José Fragoso Santos, Nuno Santos, and Pedro Adão. Concolic Execution for WebAssembly. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:29, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2022.11.
- 25 Michelson: The language of smart contracts in Tezos. URL: <https://tezos.gitlab.io/alpha/michelson.html>.
- 26 Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189, 2019. doi:10.1109/ASE.2019.00133.
- 27 Yuki Nishida, Hiromasa Saito, Ran Chen, Akira Kawata, Jun Furuse, Kohei Suenaga, and Atsushi Igarashi. HELMHOLTZ: A verifier for Tezos smart contracts based on refinement types. *New Generation Computing*, 40:507–540, 2022. doi:10.1007/s00354-022-00167-1.
- 28 Nomadic Lab. Michelson: the language of smart contracts in tezos, 2018-2023. Last accessed 17 October 2023. URL: <https://tezos.gitlab.io/michelson-reference/>.
- 29 Daejun Park, Yi Zhang, Manasvi Saxena, Philip Daian, and Grigore Roșu. A formal verification tool for Ethereum VM bytecode. In *Proceedings of the 2018 26th ACM Joint Meeting on*

3:26 Dynamic Logic for Symbolic Execution for Michelson

- European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 912–915, October 2018. doi:10.1145/3236024.3264591.
- 30 Corina S. Pasareanu. *Symbolic Execution: The Basics*, pages 5–20. Springer International Publishing, Cham, 2020. doi:10.1007/978-3-031-02551-8_2.
 - 31 Dominic Steinhöfel and Reiner Hähnle. Abstract execution. In Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira, editors, *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings*, volume 11800 of *Lecture Notes in Computer Science*, pages 319–336. Springer, 2019. doi:10.1007/978-3-030-30942-8_20.
 - 32 Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Security: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 67–82, October 2018. doi:10.1145/3243734.3243780.
 - 33 Philip Wadler, Wen Kokke, and Jeremy G. Siek. Programming language foundations in Agda, August 2022. URL: <https://plfa.inf.ed.ac.uk/22.08/>.
 - 34 Conrad Watt, Petar Maksimović, Neelakantan R. Krishnaswami, and Philippa Gardner. A Program Logic for First-Order Encapsulated WebAssembly. In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, volume 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:30, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2019.9.

Dynamically Generating Callback Summaries for Enhancing Static Analysis

Steven Arzt  

Fraunhofer SIT | ATHENE – National Research Center for Applied Cybersecurity,
Darmstadt, Germany

Marc Miltenberger  

Fraunhofer SIT | ATHENE – National Research Center for Applied Cybersecurity,
Darmstadt, Germany

Julius Näumann  

TU Darmstadt | ATHENE – National Research Center for Applied Cybersecurity,
Darmstadt, Germany

Abstract

Interprocedural static analyses require a complete and precise callgraph. Since third-party libraries are responsible for large portions of the code of an app, a substantial fraction of the effort in callgraph generation is therefore spent on the library code for each app. For analyses that are oblivious to the inner workings of a library and only require the user code to be processed, the library can be replaced with a summary that allows to reconstruct the callbacks from library code back to user code. To improve performance, we propose the automatic generation and use of precise pre-computed callgraph summaries for commonly used libraries. Reflective method calls within libraries and callback-driven APIs pose further challenges for generating precise callgraphs using static analysis. Pre-computed summaries can also help analyses avoid these challenges.

We present CGMINER, an approach for automatically generating callgraph models for library code. It dynamically observes sample apps that use one or more particular target libraries. As we show, CGMINER yields more than 94% of correct edges, whereas existing work only achieves around 33% correct edges. CGMINER avoids the high false positive rate of existing tools. We show that CGMINER integrated into FlowDroid uncovers 40 % more data flows than our baseline without callback summaries.

2012 ACM Subject Classification Software and its engineering → Dynamic analysis

Keywords and phrases dynamic analysis, callback detection, java, android

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.4

Supplementary Material Software (ECOOP 2024 Artifact Evaluation approved artifact):

<https://doi.org/10.4230/DARTS.10.2.2>

Software (Source Code): <https://github.com/Fraunhofer-SIT/DynamicCallbackSummaries/>
archived at [swh:1:dir:774fc1c198c94da21f9d9dc21f9a9721c9ac233c](https://scholar.archive.org/2024/06/01/774fc1c198c94da21f9d9dc21f9a9721c9ac233c)

Funding This research work has been funded by the German Federal Ministry of Education and Research and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

1 Introduction

Static analyses are commonly used for checking software for security vulnerabilities, quality defects, privacy leaks, and other properties. The callgraph is a core data structure of interprocedural static analysis. It encodes which statements in the code call which methods. When an analysis encounters a method call, it queries the callgraph for the set of callees in which to continue the analysis. If the callgraph misses edges, the respective callees are not considered and the analysis is incomplete. If the callgraph, on the other hand, contains spurious edges, irrelevant subtrees in the callgraph must be processed. This may not only impact performance and scalability, but may also lead to false positives.

 © Steven Arzt, Marc Miltenberger, and Julius Näumann;
licensed under Creative Commons License CC-BY 4.0
38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 4; pp. 4:1–4:27



4:2 Dynamically Generating Callback Summaries for Enhancing Static Analysis

Computing a complete and precise callgraph is non-trivial due to virtual dispatch and exceptional control flow. Approaches such as SPARK [16] already handle polymorphism well. SPARK builds upon the Escape Analysis in Soot [14] to handle exceptional control flow precisely. Nevertheless, several problems remain unsolved. Firstly, software commonly [17] uses third-party libraries, which can contribute significant amounts of code. Large code bases, in turn, pose scalability challenges for callgraph generation. Secondly, libraries can be complex and, e.g., use reflective method calls, contain asynchronous control flow, or manage callbacks in collections. A callgraph analysis must be able to handle all these language features correctly. Some libraries such as the popular OKHTTP3 library have separate callbacks for successful and failed requests. The failure callback is only executed when an exception is thrown within the library code. An approach without support for exceptional flows misses these callbacks. In practice, existing callgraph analyses apply approximations that lead to spurious or missing edges, or are too complex to scale to large programs.

We observe that, from the perspective of most client analyses, only the interface of a library is relevant. Callgraph edges inside the library only need to be computed as a means to obtain edges that cross the library’s interface through callbacks. We therefore argue that these API edges can be summarized once as a one-time effort. Only the information which API calls trigger which callbacks must be retained. Intuitively, the summary is a list of such target callbacks. When a client analysis encounters a call to an API method for which a summary is available, it plugs in the summary instead of analyzing the library. More precisely, it adds a callgraph edge to each callback described in the summary. A similar reasoning has already been applied to the data flow behavior of libraries [2].

Manually assembling these summaries is inefficient, since many libraries use callbacks. Furthermore, each new version of each library would need to be studied to reflect the changes made to the library API in the model. Consequently, the generation of these callback models must be automated. Static approaches [8] share the shortcomings of static callgraph analyses and require coarse approximations, leading to false positives as we show.

Our approach CGMINER is based on dynamic analysis instead. CGMINER takes a set of programs that use libraries of interest. It statically instruments these programs such that each call to a library function reports dynamic callgraph data to an analyzer. CGMINER executes all instrumented programs and combines the resulting execution traces into a *callback summary* for the respective library. It abstracts away from all program-specific behavior and reduces the summaries to edges between methods in the public interface of the library, omitting calls within the library itself leading eventually to the execution of a callback. If a library, for example, takes a callback as the first argument on an API call and then invokes a method on the object passed as the first argument, there is an edge between this API method and its first parameter. This edge exists regardless of which program uses the API and what the concrete callback implementation is. CGMINER captures such abstract callback semantics on the API level. With its focus on dynamic analysis, CGMINER avoids many of the common challenges in static analysis such as precisely modeling reflective method calls and complex collection types.

In this work, we focus on Android apps. On Android, libraries are compiled into the apps that use them, rather than being shared between apps. Consequently, a wide variety of libraries is used in apps [26], and handling them efficiently is vital for each app analysis. Furthermore, almost 2 million apps are available in the official Play Store. Since for each library, CGMINER requires a sample set of apps that use the respective library, such a freely accessible data source is beneficial. In addition, Android is widely used for dealing with sensitive data and functions, making client analyses that depend on callgraphs for,

e.g., finding data leaks, highly relevant in practice. We show that CGMINER can identify efficiently complex callback edges. While other approaches lead to true positive rates of more than 33% with an additional 20% of edges being incomplete, CGMINER achieves more than 91% correct edges. Note that CGMINER focuses on control flow and is intended to be combined with analysis-specific summaries such as StubDroid [2] for data flow analysis. Even with a simple integration approach, CGMINER summaries lead to 28% more correct flows being discovered than without callback summaries.

The remainder of this paper is structured as follows. Section 2 contains some background information about Android. Section 3 shows a motivating example. Section 4 explains the CGMINER approach. We describe implementation details in Section 5, before explaining CGMINER’s limitations in Section 6. Our empirical research questions and evaluation data is contained in Section 7. Finally, we present related work in Section 8 and conclude in Section 9.

2 Android Background

Android applications are written predominately in Java and Kotlin. The compiler translates this code into Dalvik byte code, which is similar to Java byte code used by the JVM. After compilation, the Dalvik byte code is written into one or multiple *classes.dex* files. The dex files as well as the necessary resource files of the app are packaged in an APK file, which is ultimately just a ZIP file. Android apps may use system classes from the `java(x)` and `android` packages. The implementation of these system classes are shared between different apps and reside on the device. On the host computer, the Android SDK installs a stubbed version of this Android system classes, which only contains the method signatures and class hierarchies of the actual implementation. This stub jar is used to link against during compilation, but does not contain the actual implementation code. In contrast to the system implementation, all other third party libraries and their transitive dependencies are compiled to Dalvik byte code as well and placed alongside the application code in the same dex files, so that each application is self-contained.

3 Running Example

Listing 1 shows a program that uses a simplified API for communicating with a remote server once the app’s main activity is launched. The library is a slightly adapted version of the OKHttp library. For the sake of brevity, we omit the implementation of the library and instead explain it using the code of the example program.

The library’s main class `HttpLibrary` is responsible for communicating with the server. Each request is represented by an instance of the `HttpTask` class. Each task is scheduled for execution using the `schedule` method. Once all requests are scheduled, the program invokes the `runAll` method to run them against the remote server. Once a task is complete, i.e., the server has responded with results or an error, the respective callback is invoked. The implementation of the error callback is omitted in Listing 1 for brevity.

For demonstration purposes, we assume the following simplified implementation of the library. The constructor of `HttpTask` stores the callback that it receives as a parameter into a field in `HttpTask`. The `schedule` method adds the `HttpTask` to a list. Still, for not freezing the UI thread, the library is multi-threaded and performs the http request in the background. The `runAll` method spawns a worker thread that regularly polls the scheduled

■ Listing 1 Motivating Example Code.

```

1  ICompleted onComplete = new ICompleted(){
2      @Override
3      public void onCallback(String results){
4          Log.i("Web", "Results: " + results);
5      }
6  };
7  IHttpFailed onFailed = ...;
8  HttpTask task = new HttpTask("/api/do", onComplete, onFailed);
9  HttpLibrary lib = new HttpLibrary("http://www.company.com");
10 lib.schedule(task);
11 lib.runAll();

```

tasks, takes the task at the top of the worklist, and processes it. The worker thread sends the requests to the server, collects the results, and then invokes the callback. The callbacks are invoked on a different thread than the original call to `schedule` or `runAll`.

Callgraph algorithms traditionally do not model the delayed behavior of the callback and instead insert an edge from the call site that causes the callback to be executed to the callback implementation, e.g., from `Thread.start` to the `run()` method of the thread. CGMINER adopts the same behavior. Its callback summaries model an edge from `runAll` to the `onCallback` method of both completion and error callback. In other words, our model assumes that `runAll` immediately invokes both callbacks. The challenge in this example, which is not handled by existing approaches, arises because the callbacks are not in the scope of the call site for `runAll`. Instead, the summary generator must automatically infer the link to the `HttpTask` instances that were scheduled, and then to the actual callbacks that were passed to the constructor of the `HttpTask`.

Existing approaches such as EdgeMiner [8] model the callgraph edges in Line 8. This reduces the complexity of the example, because the callback is not passed across multiple classes. On the other hand, such a model is incompatible with a flow-sensitive analysis. To illustrate this, we change the example slightly as shown in Listing 2. For this example, suppose that the `source` method call in Line 9 returns sensitive information. Further assume that the parameter of the `sink` method (called in Line 4) is sent to a remote server. This example uses the field `data` to save the sensitive information, which is leaked in the callback method.

Consider we are running a flow-sensitive taint analysis such as FlowDroid [5] to detect this data flow from `source` to `sink`. FlowDroid starts at the source [4] statement in Line 9 and advances forward through the control-flow graph until it reaches a sink. It reports a leak when the sink is reached with a tainted parameter. During the propagation, it keeps track of all variables that may contain sensitive information (“tainted”).

Notice that the callback is passed to the library in Line 8, before the `source` method is called. During execution, the call to `runAll` in Line 12 invokes the callback which leaks the data. Nonetheless, EdgeMiner inserts an edge at Line 8 to the `onCallback` method. Because the call to `source` happens after the callback registration, FlowDroid does not encounter the sink statement when using the EdgeMiner summary. Consequently, the leak is missed. In this example, FlowDroid needs an edge from the `runAll` call in Line 12 to the callback method in order to reach the sink and thus detect the dataflow.

Listing 2 Flow-Sensitivity Example.

```

1 ICompleted onComplete = new ICompleted(){
2     @Override
3     public void onCallback(String results){
4         sink(data);
5     }
6 }
7 IHttpFailed = ...;
8 HttpTask task = new HttpTask("/api/do", onComplete, onFailed);
9 data = source();
10 HttpLibrary lib = new HttpLibrary("http://www.company.com");
11 lib.schedule(task);
12 lib.runAll();

```

Such approximations as in existing work have even greater negative consequences. In yet another modification of the example, imagine that the `runAll` method never calls `onFailed`, but throws an exception instead. The `onFailed` callback only exists for a second method `tryRun` that calls `onFailed` in the case of an error and that never throws an exception. In that case, EdgeMiner would still model the edge from the `HttpTask` constructor call in Line 8 to `onFailed`. This edge is clearly invalid if the program never calls `tryRun`. The EdgeMiner model does not contain any notion of `runAll` and `tryRun` and, hence, cannot make this distinction.

4 Approach

To avoid the inherent challenges of static analysis described in Section 1, CGMINER relies on dynamic analysis for inferring the callgraph summaries on libraries. Figure 1 shows the architecture of the analysis. CGMINER takes as input the original APK file and a list of classes that correspond to libraries. These apps are then instrumented with three analyses: a general-purpose dynamic callgraph analysis, a general-purpose dynamic taint analysis and a specialized callback analysis into the app. Since libraries (except for the Android Framework) are part of the application code in Android, library code can be instrumented as well.

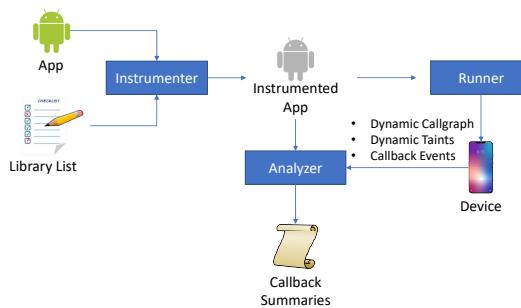


Figure 1 CGMINER Approach. The app is instrumented and then executed on a device, with runtime events being routed to the callgraph analysis. Events include dynamic callgraph, dynamic taint tracking, and specific callback analysis events.

The instrumented app is passed to the runner, which installs it on a device, and establishes a communication channel with the app. It forwards all events sent by the analysis code injected into the app to the analyzer. The analyzer is responsible for processing the events

and for inferring the callback summaries while the app is running. In the remainder of this section, we explain how the analyzer derives the callback summaries and how the different components (dynamic callgraph, dynamic taint analysis, callback analyzer) interact.

4.1 General Idea

CGMINER needs to identify which statements trigger which callbacks. The app is instrumented with event tracing that reports back to the analysis computer whenever a potential callback method is invoked on the phone. When such a method is invoked, CGMINER uses a combination of dynamic control flow analysis and dynamic taint tracking to identify all API calls between the point where the callback class was originally passed to the library (constructor of `HttpTask` in the example) and the callback method.

When arriving inside a callback, CGMINER must find the API call that triggered the callback (method `runAll` in the example). Intuitively, this can be done by searching backwards through the dynamic control flow graph. This approach, however, does not identify the necessary state changes to the `HttpLibrary` object. Recall from Section 3 that the `runAll` method would not invoke any callback unless the callback has previously been registered with the library using the `schedule` method. In other words, the inner state of the `HttpTask` object must be changed before calling `runAll`. More generally, the callback is passed through several objects along the way, and CGMINER must identify the API calls that lead to these state changes, i.e., that copy the callback around. In the example, the constructor of `HttpTask` assigns the callback to a field and the `schedule` method adds the task to the list, which is finally processed by `runAll` (see Section 3). We call such methods *transfer methods*. The user code must call this method for the callback to be registered, and it must therefore be part of the callback summary.

Intuitively, when an API method stores a callback in a field or collection inside of some object, the transfer method is the last API call that transitively lead to the assignment. To find the relevant assignments, CGMINER relies on dynamic data flow analysis. CGMINER taints each callback object when it is first passed to an API call (constructor of `HttpTask` in the example), i.e., each callback object is considered a source for the dynamic taint tracking algorithm. It then follows this taint through the library, until it arrives at the `this` object inside a callback method. The start of a potential callback method is considered a sink.

The taint state on the device is always mirrored to the analysis computer. When the analyzer observes that a callback has been called, it retrieves the taint paths, i.e., all statements that have passed the taint from one object to another on the path between the callback registration and the invocation of the callback method. Whenever a new object is tainted, e.g., through an assignment to a field, CGMINER searches the dynamic callgraph backwards to find the API call that triggered this taint transfer, i.e., the methods that user code must call before `runAll`.

4.2 Overview of the Approach

We define a *border edge* as an edge that is from library code to user code or vice versa. In other words, the caller is in library code and the callee is user code (i.e., a callback), or the caller is in user code and the callee is library code (traditional call to a library method). Edges in the first case, i.e., callbacks, are denoted *out* edges, whereas edges in the second case, i.e., normal library calls, are *in* edges. Figure 2 shows an abbreviated control flow graph for Listing 1. It further shows the *in* and *out* edges for the motivating sample. In the example, the `HttpTask` and `HttpLibrary` constructor calls and the calls to `schedule` and

`runAll` are *in* edges, because these calls in the application code directly call library methods. The `runAll` method iterates over all scheduled tasks, which were registered in the `schedule` method, and calls a library-internal `run` method. The `run` method calls the `onComplete` on the callback. Thus, this method call constitutes an *out* edge.

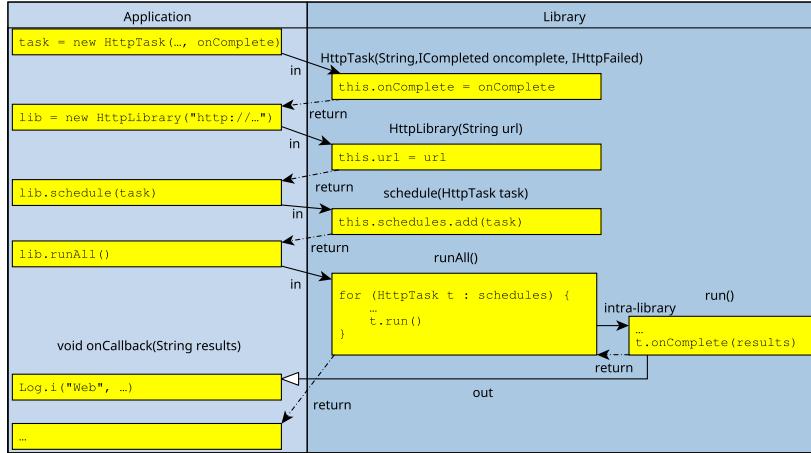


Figure 2 Shows *in* and *out* edges for Listing 1. *in* edges are regular arrows. The *out* edge is denoted by a white tip.

As a pre-analysis, CGMINER statically over-approximates the potential callback implementations, i.e., potential targets of *out* edges. We call these methods *callback candidates*. Afterwards, CGMINER statically approximates a set of all possible callback classes. In Listing 1, the constructor `HttpTask` may register a callback. All classes that (transitively) implement the interfaces `ICompleted` and `IHttpFailed` are potential callbacks. Specifically, we have the anonymous inner class implementing `ICompleted` and the (omitted) corresponding inner class for `IHttpFailed`. These classes override the interface methods, in this case only `onCallback`. These overridden methods represent the potential targets of *out* edges. These steps for identifying callback registration sites and potential callbacks are static over-approximations, which are used to bootstrap the dynamic analysis that follows later.

The dynamic analysis is used to determine which callback candidates are reached and to perform dynamic taint tracking in order to track which API calls (such as the `schedule` and the constructor calls in Listing 1) are necessary.

At the start of each callback candidate, we statically instrument a call to a method we call *reporting method*. This reporting method sends an event with information about the triggered callback to the analyzer. Because irrelevant callback candidates or spurious callback implementations will later not be reached during dynamic analysis, they will not become false positives. We provide details on how potential callbacks are identified in Section 4.3.

When a callback is invoked at runtime and the respective callback event is triggered, the analyzer uses the dynamic callgraph to find the corresponding *in* edge. The call site at the *in* edge represents the API method that triggers the callback. Note that the analyzer skips the library-internal calls between *in* and *out* edge. The *in* and *out* edges are trivial to identify based on the classes in which the respective calls and their corresponding callees are located. In the example from Listing 1, the analyzer deduces that a call to `HttpLibrary.runAll` invokes `IHttpCompleted.onCallback`. Further, CGMINER uses dynamic taint tracking to find the transfer methods. Therefore, the statements at the beginning of all potential callback

methods are marked as sinks for the dynamic taint analysis. To build the list of sources, CGMINER first collects all API sites that receive instances of potential callback classes as arguments. These API calls are then registered as sources in the dynamic taint analysis such that the respective potential callback classes, i.e., the call arguments, are tracked at runtime. Note that this approach is an over-approximation. When generating the callback summaries, CGMINER only relies on the taint paths that were actually taken at runtime. Therefore, marking too many classes as potential callbacks or registering too many APIs as sources does not reduce CGMINER’s precision. CGMINER assigns an unique ID number to each taint source in order to distinguish different sources.

Transfer methods do not need to operate on the original heap object. The `schedule` method in Listing 1 never touches the callback object. It operates on the `HttpTask` that encapsulates the callback. An approach based on object identity alone would miss the `schedule` method. Dynamic taint tracking, on the other hand, can taint the `HttpTask` object when it encounters the assignment inside the constructor of `HttpTask`. Recall that this constructor receives the callback and stores it in a field. Afterwards, the dynamic taint tracker follows the `HttpTask` object as well. Similar reasoning must be applied for the `schedule` method, which stores the `HttpTask` object in a list, i.e., the list must be tainted as well.

With this taint tracking information, CGMINER can identify all border edges by looking at the taint transfers. Recall that the first statement inside the callback is a sink. CGMINER can query the taint analysis for the corresponding source and all statements in between at which taint was transferred to fields or collections (details in Section 4.5). For each statement on the taint path, it queries the dynamic callgraph to identify the corresponding *in* edge, i.e., the call from the user code that lead to the taint transfer. In the example, this allows CGMINER to identify the call to the constructor of `HttpTask` (transfer: field assignment) and to `schedule` (transfer: collection). Approaches that only inspect the call chain that ends at the callback (`runAll` in the example) would miss these intermediate calls.

4.3 Identifying Potential Callbacks

Recall that the callback analysis is started when a callback is invoked. Consequently, each possible callback method must be instrumented with an event that notifies the analyzer about which method has been called at runtime. This section shows the static analysis phase of the CGMINER approach. In this phase, CGMINER first determines possible callbacks and then instruments code in order to be notified when a callback happens.

To find the potential callbacks, CGMINER identifies all statements in the app that call library methods. The approach then checks whether a reference to a callback object is passed as an argument. We define a callback argument as follows. The declared type of the respective parameter is a reference type from the library, i.e., a class or an interface, which must be non-final and accessible to user code according to its access modifiers. Further, if the library class is a class and not an interface, it must have a non-private constructor. The type of the argument that is passed must be a class type from the user code that (potentially indirectly) inherits from the library class or implements the library interface. This class type represents a potential callback implementation. The type of the callback class can be approximated statically, either by identifying the allocation site at which the callback object was instantiated, or by looking at the declared type of the callback variable that is passed as an argument. To be complete, CGMINER applies a Variable Type Analysis (VTA) style analysis [24] to identify all possible types based on the declared type if no precise allocation site is available. Considering too many potential callbacks only leads to more instrumentation effort and does not affect the precision of CGMINER, as these spurious callbacks are not triggered at runtime.

CGMINER only identifies a set of potential library classes in this step. The concrete library method to which the callback object is passed is irrelevant for the analysis in this stage and is discarded.

Potential callbacks must be instrumented. Since this is not possible for Android and Java system classes, CGMINER automatically wraps these classes. For example, when `java.lang.ArrayList` is instantiated in the application code, CGMINER replaces the call so that a wrapped version of `ArrayList` is called, which can then be instrumented. The wrapped variant inherits from `ArrayList` and forwards all protected and public methods to their corresponding super class implementations. CGMINER not only wraps constructor calls, but also values returned by system classes, e.g. `ArrayList.iterator()`.

4.4 Dynamic Callgraph Analysis

For building the dynamic call graph, CGMINER instruments the app as follows. Before each call, a `CALL` event is sent from the device to the analyzer with the unique ID of the call site. After the call site, i.e., when the call has returned, a `RETURN` event is sent for the same ID. At the beginning of each method, a `ENTER` event is sent. Before each `return` or `throw` statement, a `LEAVE` event is sent. These events allow the analyzer to reconstruct the call edges taken on the device. Due to memory and performance constraints, CGMINER does not build the dynamic callgraph on the device. Once the events are sent, they are immediately discarded on the device.

The analyzer maintains a separate call stack for each thread. For each `CALL` event, the respective call site is put on the stack. When the analyzer receives an `ENTER` event, it creates a call edge from the top call site on the stack to the method that was entered. Note that a `CALL` event may be followed by multiple pairs of `ENTER` and `LEAVE` events before the `RETURN` event occurs. The Dalvik / Java runtime calls the static initializer of a class when the class is loaded. Therefore, each call may first invoke the static initializer before the actual callee is called. CGMINER captures this semantic by leaving the call site of the static initializer on the stack until the `RETURN` event is encountered. CGMINER does not consider implicit calls to static initializers from statements that are not call sites, e.g., from assignments to static variables. In this case, the static initializer generates an `ENTER` event, but has no matching call site on the stack. Therefore, the `ENTER` event is discarded. As CGMINER reconstructs call chains to callbacks, non-call initializations are not relevant.

4.5 Dynamic Taint Analysis

As explained in Section 4.2, CGMINER uses dynamic taint tracking to track the callback object through the program, including all container objects that hold this callback object. Note that only heap objects are tracked, no primitives. Therefore, the runtime code that gets instrumented into the app can uniquely distinguish each tainted object by its identity hash code (`System.identityHashCode()`). The instrumented runtime code stores a map between the unique ID of the taint source and the identity hash code¹, and also transmits this map to the analyzer on every change, i.e., whenever a new object is tainted. These transmissions occur on the background based on a transmission queue. The events are sent as one message whenever a certain number of events have accumulated. Therefore, the transmissions do not affect the performance of the original app.

¹ The implementation takes care of checking referential equality in case of hash collisions.

All field assignments are instrumented to perform *taint transfers*. If a variable is assigned to a field, the runtime code checks whether the variable on the right side of the assignment is tainted, i.e., its identity hash code is in the taint map. If so, the base object that contains the field is tainted as well, i.e., its identity hash code is written into the map with the same source ID as the variable. These *derived taints* are field-insensitive by design. If a library stores two different callbacks in two fields of the same object, this object is associated with both sources.

Recall that the dynamic taint tracking in CGMINER is special, since the object that is tainted at the source (the callback object) is always the same object that arrives at the sink (the `this` object inside the callback). The taint tracking is only used to track the path between where the callback was registered in the library and where the control flow arrives inside the callback. This allows for some imprecision in the taint tracking, because flows where source and sink object are not identical can be discarded.

The Java Standard Library cannot be instrumented, because it is pre-installed on the device and not part of the app. For such cases, CGMINER relies on the static taint data flow summaries from StubDroid [2]. Based on these summaries, CGMINER adds instrumentation at the call site in the user code rather than instrumenting the library itself.

4.6 Callback Summary Modelling

In this section, we describe the model that we use for the callback summaries throughout the rest of the paper. Note that this section only contains the general principle of a summary. We will use this model in Section 4.7, where we describe the algorithm for generating the callback summaries.

In the simplest case, we model callback summaries as a single “fake” call edge $a \rightarrow \langle b, c \rangle$ from an API call a to a callback method b . The target is a pair $\langle b, c \rangle$, where c describes the object on which method b is called at the call site a . Recall that applying a callback summary corresponds to a virtual dispatch in the context of the original API call. In other words, the target method is called either on the same base object as the original API method, or an object passed to the original API call as a parameter. As an example, consider `AsyncTask`. `AsyncTask` is a class that is commonly used in Android, which is part of Android’s standard library, which is automatically available to every app. It is used to perform an action asynchronously, similar to a Java thread. Developers extend the `AsyncTask` class and override the `doInBackground` method, which is the callback method called in a background thread. In order to start the task, developers invoke the `execute` method on their `AsyncTask` instance. In the case of the `AsyncTask` class, the summary would be `AsyncTask.execute → ⟨AsyncTask.doInBackground, -1⟩`. The special value $c = -1$ stands for the base object of the call to `execute`, i.e., the `AsyncTask` object itself. A $c \geq 0$ would denote the c th parameter object of the caller statement using zero-indexing. Note that there may be more than one edge that originates in the same API method a . In the example, the Android OS also calls methods such as `onPostExecute` and `onPreExecute`, each of which is modelled as a separate summary edge.

The case from the example in Listing 1 is more complex, since the callback object is passed through multiple intermediate API calls, i.e. transfer functions. When applying the callback summary in a callgraph algorithm, i.e., when identifying the method that shall receive calls to `HttpLibrary.runAll`, the intermediate edges are processed in reverse order. The method `schedule` is called on the base object (index -1) of the previous call to `runAll`. The constructor of `HttpTask`, is called on the object that was the first argument (index 0) on the previous call to `schedule`. The original callback method `onCallback` is invoked on the object that was the second argument (index 1) in the previous call to the constructor of `HttpTask`.

In the callback summary, these intermediate calls are modelled as intermediate edges: $\langle \text{HttpLibrary.runAll}, -1 \rangle \rightarrow \langle \text{HttpLibrary.schedule}, -1 \rangle \rightarrow \langle \text{HttpTask.cons}, 0 \rangle \rightarrow \langle \text{IHttpCompleted.onCallback}, 1 \rangle$.

4.7 Callback Reconstruction

Algorithm 1 shows the details of how callback summaries are created. Function `BuildCallbackSummaries` is the main entry point that builds the callback summaries. It is called when the analyzer receives an event that callback method m has been called at runtime.

`BuildCallbackSummaries` uses the method `GetLastLibraryCallSite` to get the last library call site. `GetLastLibraryCallSite` in turn uses a helper method `GetDynamicTraces`, which returns a set of call traces that end in statement s by performing a graph search on the dynamic callgraph. The call graph is flattened into a set T of sequences of call sites c . Recursions are unrolled once in `GetDynamicTraces`, since repeating the same sub-sequence of calls does not provide any additional insights for the purpose of callgraph analysis. For not bloating the description, we do not present the implementation of the helper method `GetDynamicTraces` in the pseudocode.

Given a statement inside library code, method `GetLastUserCodeCallSite` uses the traces returned by `GetDynamicTraces` and returns the last user code statement that happened before and that transitively triggered the given library code statement. Similarly, method `GetLastUserCodeCallSite` takes a statement inside a callback in user code and identifies the last library statement that happened before and (transitively) invoked the given statement from the callback method. The helper method `Predecessor` takes a statement and returns the predecessor statement on the dynamic callgraph. For simplifying the presentation, we assume that this statement is unique. Our implementation can handle multiple candidates.

The main summary generator `BuildCallbackSummaries` first obtains the last statement in the library code before the callback was invoked (line 20). This is the last statement in the library before the control flow is passed back to user code, i.e., the *out* edge. The relevant interactions between user code and library API occur between this statement and the statement that originally passed the callback to the library (the *in* edge and source for the dynamic data flow analysis). We will explain the special case of $S_c = \epsilon$ (first branch in line 21) later. In line 25, CGMINER uses the method `GetPathsBetween` to query the dynamic taint graph for all taint paths between the two statements. A taint path is a sequence of statements that assigns a tainted variable or field to another variable or field, i.e., passes around a reference to a tainted object. This definition implies that method calls are part of the taint path as well, because they assign the value of the tainted argument at the call site to the corresponding parameter variable inside the callee. Note that there can be more than one path between source and sink, so \mathbb{P} is a set of lists of statements. CGMINER first iterates over all paths and then over the statements in each path. It builds a new summary for each path. Hence, the initialization of the summary (line 27) is inside the loop over the paths.

The summary starts with the statement that passes the callback object to the library, i.e., the *in* edge. This statement is simply the source from which taints arrive in the callback method, as shown in line 27. Method `GetSource` returns the API at which the source was registered. The assignment statements on the taint path that copy around the callback object inside the library are not directly visible to the user code. Instead, the user code calls API methods that transitively trigger these statements through library-internal call chains. In the example, an assignment somewhere inside `schedule` or one of its transitive callees assigns the parameter with the task to a field. This assignment is on the taint path, but only the preceding call to the transfer method `schedule` is relevant as a part

■ **Algorithm 1** Callback Reconstruction Algorithm.

```

1 Function GetLastUserCallSite( $s$ ):
    INPUT:  $s$  – the first statement in the callback
    OUTPUT: The last statement in user code

     $T = \text{GetDynamicTraces } (s)$ 
    foreach  $t \in T$  do
        foreach  $c \in t$  do
             $c' = \text{Predecessor } (c)$ 
            if not IsLibrary ( $\text{GetMethod } (c')$ ) then
                if IsLibrary ( $\text{GetMethod } (c)$ ) then
                    return  $c'$ 
    return  $\epsilon$ 
9

10 Function GetLastLibraryCallSite( $s$ ):
    INPUT:  $s$  – the first statement in the callback
    OUTPUT: The last statement in library code

     $T = \text{GetDynamicTraces } (s)$ 
    foreach  $t \in T$  do
        foreach  $c \in t$  do
             $c' = \text{Predecessor } (c)$ 
            if not IsLibrary ( $\text{GetMethod } (c)$ ) then
                if IsLibrary ( $\text{GetMethod } (c')$ ) then
                    return  $c'$ 
    return  $\epsilon$ 
18

19 Function BuildCallbackSummaries( $m$ ):
    INPUT:  $m$  – the callback method
    OUTPUT: A set of callback summaries

     $\Delta = \emptyset$ 
20     $S_c = \text{GetLastLibraryCallSite } (\text{FirstStmt } (m))$ 
21    if  $S_c = \epsilon$  then
22         $\phi = \text{GetSource } (m)$ 
23         $\Delta = \{\phi \rightarrow \langle m, \gamma(m) \rangle\}$ 
24    else
25         $\mathbb{P} = \text{GetPathsBetween } (\text{GetSource } (m), S_c)$ 
26        foreach  $p \in \mathbb{P}$  do
27             $\delta = \omega(m)$ 
28            foreach  $s \in p$  do
29                 $S_u = \text{GetLastUserCallSite } (s)$ 
30                 $\delta = \delta \circ \langle \omega(S_u), \gamma(S_u) \rangle$ 
31             $\Delta = \Delta \cup \{\delta\}$ 
32    return  $\Delta$ 
33

```

of the summary. In line 29, CGMINER uses the helper method `GetLastUserCallSite` to identify this corresponding API method by conducting a backward search in the dynamic callgraph. For statements that are already in user code, i.e., the first statement on the path, `GetLastUserCallSite` is an identity function.

Each identified transfer statement in user code maps to one fragment of a summary. In line 30 the current API method is concatenated to the summary built so far. For example, if the summary $\text{HttpLibrary.runAll} \rightarrow \langle \text{HttpLibrary.schedule}, -1 \rangle$ existed before, a new right arrow is appended to the next method and parameter index. As explained above, for the last statement of a taint path, $S_u = S_c$ holds, i.e., a taint path always ends with the API call that finally invokes the callback. S_c is the last library call site (line 20 in Algorithm 1), i.e., the last statement that was executed in the library before invoking the callback.

For extending the summary, CGMINER uses two helper functions: ω and γ . The ω method performs the generalization from concrete statements and methods to API interfaces. For call sites, ω retrieves the API signature. For callback methods, ω retrieves the name of the interface or abstract API class that declares the method. The γ method takes a call statement and identifies the tainted parameter, i.e., the parameter that contains the callback object, by querying the dynamic data flow graph. As explained in section 4.6, CGMINER uses the special value -1 if the base object of the call is tainted.

Note that Algorithm 1 also works for cases without transfer methods. Android's `AsyncTask.execute` method is part of the Android SDK, i.e., a pre-installed library on the device. It cannot be instrumented. Conceptually, the *in* edge points to a fake node (a method for which we have no implementation) and the *out* edge points from this node to the callback method `doInBackground`. In this case, the summary is a simple edge from a single API call site to a single callee method as explained in Section 4.6. In the algorithm, $S_c = \epsilon$ holds, and the first branch is taken in line 21. CGMINER retrieves the taint source, i.e., the *in* edge and construct an edge to the callback method m . The parameter index is derived from the taint graph using an overload of γ that processes the parameter variables of m instead of the call arguments at a call site.

4.8 Extensions and Special Cases

For simplicity, the algorithm presented in the pseudocode assumes that a single callback method is only connected to a single source, i.e., `GetSource` returns a single method. In other words, the developer does not re-use the same callback implementation for different independent API calls. Our implementation supports such re-uses.

Further, recall from Section 4.5 that CGMINER uses StubDroid summaries to model the effects of methods that cannot be instrumented in the dynamic taint analysis. In these cases, the transfer method cannot be found using a backwards search on the dynamic callgraph as shown in line 29. Instead, CGMINER marks these statements and directly uses statement s in such a case.

4.9 Applying Summaries

Many static analysis approaches require a callgraph. Computing the callgraph on the application code as well as the code of all libraries required by the application can require significant computational resources and time. Therefore, it makes sense to replace the libraries by summaries. These summaries must capture the control flow of the library with respect to its external interface, i.e., it must correctly model callbacks back to the application code. CGMINER generates such summaries. They can be applied whenever a callgraph is needed on an application that uses a library for which a summary was previously computed.

As such, we want to apply the generated callback summaries during the callback construction. In the case of the motivating sample in Listing 1, we want to apply the edge summary $\text{HttpLibrary.runAll} \rightarrow \langle \text{HttpLibrary.schedule}, -1 \rangle \rightarrow \langle \text{HttpTask.cons}, 0 \rangle \rightarrow \langle \text{IHttpCompleted.onCallback}, 1 \rangle$. In this section, we introduce Algorithm 2. In the case of the motivating example, the algorithm outputs an edge from `runAll` to the anonymous implementation referenced by `onComplete` and `onFailed`, resulting in a precise callgraph. This shows that we need the intermediate edges in order to determine the link between the implementation supplied at the constructor call (referenced by `onComplete` in the sample) and the call to `runAll`. Without these intermediate edges we have no information on the actual type of the callback object at the callback invocation site. Without such information, we would need to create edges from `runAll` to all possible implementations of `ICompleted`, even if they are not used as a callback.

Given a call site s and a set of callback summaries Δ , method `FindReceivers` in Algorithm 2 enumerates the potential callees at s . `FindReceivers` performs a traditional callgraph search via `QueryCallgraph`. It then augments these callees with the callbacks that are found by applying the callback summaries.

Algorithm 2 Summary Application Algorithm.

```

1 Function FindReceivers( $s, \Delta$ ):
    INPUT:  $s$  – the call site for which to find the receivers,
     $\delta := \langle \alpha_1, \beta_1 \rangle \rightarrow \dots \rightarrow \langle \alpha_n, \beta_n \rangle$ 
    – the callback summaries
    OUTPUT: The potential callees for the given call site

2    $\mathbb{R} = \{ \text{QueryCallgraph}(s) \}$ 
3   foreach  $(\delta := (\omega(s) \rightarrow \dots \rightarrow \langle \alpha_n, \beta_n \rangle)) \in \Delta$  do
4      $\hat{\delta} := \langle \alpha_1, \beta_1 \rangle \rightarrow \dots \rightarrow \langle \alpha_1, \beta_1 \rangle$ 
5      $\mathbb{R} = \mathbb{R} \cup \text{ApplySummary}(s, \hat{\delta}, -1)$ 
6   return  $\mathbb{R}$ 
7

8 Function ApplySummary( $s, \delta, i$ ):
    INPUT:  $s$  – the call site for which to find the receivers,  $\delta$  – the summary,  $i$  – the
    argument index
    OUTPUT: The potential callees for the given call site

9    $\hat{\delta} := \langle \alpha_2, \beta_2 \rangle \rightarrow \dots \rightarrow \langle \alpha_n, \beta_n \rangle$ 
10   $v = \text{VariableOf}(s, i)$ 
11   $S = \text{GetCallsOn}(v)$ 
12   $\mathbb{R} = \emptyset$ 
13  foreach  $\hat{s} \in S$  do
14    if  $\hat{\delta} = \epsilon$  then
15       $\mathbb{R} = \mathbb{R} \cup \{\kappa(\hat{s}, v)\}$ 
16    else
17       $\mathbb{R} = \text{FindReceivers}(\hat{s}, \hat{\delta}, \gamma(\hat{s}, v)))$ 
18

```

For applying the callback summaries, line 3 iterates over all summaries δ in the database Δ . It looks for those summaries that reference the API call from the given statement. Recall from Section 4.7 that $\omega(s)$ extracts the generic API method signature from a concrete statement. Each summary is applied using method `ApplySummary`. Note that the source statement $\omega(s)$ is removed from the sequence of calls inside the summary and only the remaining calls are passed. Method `ApplySummary` processes these intermediate calls recursively and removes one call per iteration until the final call, i.e., the one that invokes the callback method, is found. We included the structure of δ in line 1 for clarity. Line 9 shows the derived $\hat{\delta}$ with the first element removed from the summary.

For the structure of the individual calls on the summary, recall from Section 4.6 that the first element a encodes the API method, and b encodes the base object on which the API method is called. $b = -1$ refers to a call on the base object, $b \geq 0$ references the parameter with the respective index.

Method `VariableOf` in line 10 obtains the variable v that corresponds to index i in the context of statement s . CGMINER then obtains all virtual call sites $\hat{s} \in \mathbb{S}$ where variable v is the base object using method `GetCallsOn`. For each of these call sites, CGMINER continues the search for the element of the call summary using a recursive call to `ApplySummary` in line 16. Method γ takes a statement, which must be a call site, and a variable, and returns the index of that variable in the argument list of the call (or -1 if the variable is the variable is the base object of the call).

The recursion ends if the summary has no further calls to analyze (line 14). Method κ takes a statement and a variable, e.g., `s.onCallback()` and s . It returns the method that is called (`onCallback` in this case), which is the final callee that is added to the callgraph. Note that `ApplySummary` calls `FindReceivers` again once the statement and variable of the callback are known. This is necessary, because callbacks usually rely on virtual dispatch, i.e., the actual receiver depends on the possible types of the base object. In the example from Listing 1, multiple classes could implement `IHttpCompleted` and depending on some conditional, variable `completedCallback` could be any one of them at line 11. CGMINER detects that `completedCallback.onCallback` is line 11. Finding the final receivers of this virtual call is an orthogonal problem and CGMINER relies on the existing callgraph algorithm.

CGMINER only summarizes callgraph data and must be extended with summaries that capture the semantics of the client analysis. CGMINER integrates well with StubDroid [2], which summarizes data flow, but does not address control flow.

5 Implementation

We run the sample apps on real devices using DFarm [19]. For instrumenting the code and interacting with the devices, we rely on the VUSC commercial code analyzer. VUSC provides an API for instrumenting value requests into Jimple [25] code and for associating the events received at runtime with the Jimple statements at which they were generated. The device communication is derived from FuzzDroid [23] and uses Soot for instrumentation [3].

The runtime overhead of the additional code injected by CGMINER is not relevant as long as the app does not crash with an Application Not Responsive (ANR) exception. The Android system automatically sends ANRs when foreground threads (such as the UI thread) are blocked for an extended amount of time. In order to avoid ANRs, we queue events in the corresponding thread in which they occur and sent them asynchronously. The communication with the control computer happens in a separate thread controlled by an Android Service.

For the list of library classes that serves as an input to CGMINER, we crawled the Maven central repository as well as the Google Gradle repository. To limit the size of the database, we only include libraries that are referenced by at least five other Maven artifacts. For these relevant libraries, we extract the package names of all classes contained in the respective JAR file. When running CGMINER, we consider a class to be a library class when it is contained in one of these known library packages. Note that library identification is orthogonal to the callback analysis, and CGMINER is agnostic to how the list of library classes is built.

6 Limitations

CGMINER instruments the Dalvik code inside an app. If parts of the control flow between API call and callback are implemented in native code, no runtime data can be obtained from these parts. If the native code contains border edges, the callback summary will be incomplete. If a taint transfer occurs in native code, CGMINER relies on StubDroid summaries, which exist for methods from the Java Standard Library, such as `System.arraycopy`.

Note that Android requires each APK file to be signed. Therefore, when instrumenting an app, the app needs to be resigned. Since CGMINER modifies the app for the dynamic analysis, it must be signed anew before it can be installed on the device. If the app performs integrity checking, these checks will fail. While the individual app cannot be analyzed in this case, the CGMINER approach still works if, for each library, enough apps that use the respective library can be processed.

Not every callback may be invoked in each run of each app. In our example in Listing 1, the error callback is only invoked if the HTTP connection fails. Since the callback summaries are merged over many apps in CGMINER, we consider edges that are never triggered even with dozens of apps to be irrelevant in practice.

Our evaluation is partly based on Monkey [10] for exploring the apps' user interface. Monkey is part of the official Android SDK and randomly clicks on the screen for a given amount of time. Note that CGMINER is agnostic to the input generation tool. It can be replaced with a more capable approach in future work. We also used manual exploration in order to augment the automatic analysis.

We currently do not consider Android lifecycle methods such as `onCreate`, as they are few, well-known, change rarely, and are already precisely modeled, e.g., in FlowDroid [5].

7 Evaluation

In this section, we evaluate CGMINER with regard to the following research questions:

- RQ1** How many callback edges does CGMINER identify?
- RQ2** Are the callback summaries correct and complete?
- RQ3** How long does the instrumentation take?
- RQ4** How often do transfer functions occur?
- RQ5** How does CGMINER compare to EdgeMiner?
- RQ6** Which summaries have been found (case study)?
- RQ7** How do summaries affect data flow analysis?

7.1 Experiment Setup

We used a machine with 144 Intel Xeon Gold 6154 CPU cores and 3 TB of physical memory using OpenJDK 16. A maximum of 50 GB was assigned as Java heap space. The machine was chosen due to the performance requirements of FlowDroid for RQ6. Our DFarm installation

is equipped with around 90 devices in total, comprising Samsung Galaxy XCover Pro phones distributed over 9 device controller boards and a single master controller. Note that each run of CGMINER only uses a single device. We use a combination of manual and automatic exploration. For automatic exploration, we used the Monkey tool from the Android SDK to explore the user interface of the app at runtime. Despite its simplistic approach, Monkey achieves code coverage results comparable to more complex approaches [9]. We run each app for five minutes with automatic and the same time using manual exploration. In apps where a login was needed in order to proceed the exploration of the app, we manually created user accounts.

For our callback generation, we randomly collected 700 apps from the Google Play Store between 2008 and 2021, augmented with apps from AndroZoo [1]. We include older apps to merge the callback summaries over multiple versions of a library, and to also include error cases, e.g., failing HTTP connections due to the server no longer being operational. In our experience, newer versions of libraries return the old methods (including their callbacks) for backwards compatibility. On the other hand, new versions may introduce new additional methods with callbacks, requiring us to run CGMINER again on the new version.

For research questions 2 and 5, we inspected callback summaries manually. Two researchers conducted the manual inspection. Upon disagreeing, a third researcher has been involved and these cases were discussed until a consensus was reached.

7.2 Baseline over the Dataset

To better understand the performance of CGMINER, we measure the sizes of the original apps in our dataset, i.e., before the instrumentation. The number of classes ranges from 6 to 37,175 with an average of 14,371 and a median of 13,136. The apps contain between 108 and 226,966 methods, with an average of 85,270 and a median of 69,360 methods. In the Jimple intermediate representation, the apps contain between 693 and 2,793,272 units (i.e., Jimple instructions), with an average of 1,107,276 and a median of 1,008,848 units.

7.3 RQ1: Number of Generated Callbacks

From the 700 apps in our dataset, CGMINER created callback summaries for 338 apps. Not all apps contain callback-driven libraries according to our definition. Hybrid apps, for example, implement their logic in JavaScript and only present HTML content to the user via Android's `WebView` component. Other apps use libraries that our library detection does not recognize, or simply do not use callbacks. Some apps contain native code, which is not supported in CGMINER. Recall that CGMINER creates summaries for libraries rather than apps. Furthermore, some apps are merely add-ons such as themes for other applications and do not have any launchable main activity. Therefore, as long as a single app uses the library's callback-driven API, a summary can be generated.

In total, CGMINER constructed 1,476 summaries, which is around 8 summaries per app on average. Figure 3 shows the cumulative distribution of the number of callbacks found per app. The x axis is the number of edges, and the y axis shows how many apps lead to the given number of edges. The maximum number of edges obtained from one app is 67, the minimum is zero, with a median of 5. For each summary, we recorded the number of API methods that must be called to invoke the callback. On average, one call is required, with a maximum of 2 calls and a minimum of one call. The median is one call.

To augment our callback summaries, we automatically generated artificial apps in an attempt to trigger the callback candidates for which CGMINER did not yield an edge on our original app set. This is a best effort approach. We accept that some of these apps will

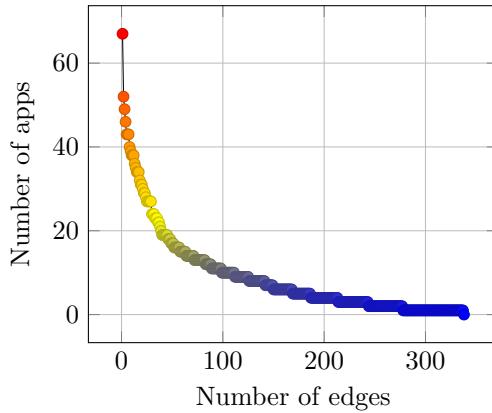


Figure 3 Distribution of callback edge counts over the apps.

crash or fail to invoke the callback. Recall that callback candidates are over-approximated, i.e., it may be impossible to generate a working app for some candidates. However, since CGMINER is a dynamic approach, broken apps do not lead to false positives in the callback summaries generated by observing these apps. The generated artificial apps yielded 1,871 edges.

7.4 RQ2: Correctness of Generated Callbacks

We manually verified the callback summaries generated by CGMINER. We merged the summaries from apps in the dataset with the generated apps mentioned in RQ1. We found 94.62% of all callback summaries to be correct. 38 edges out of 2046 were false positives. Transfer statements were missing in 72 cases.

Since CGMINER is a dynamic approach, it is inherently an underapproximation. To better understand the degree of unsoundness, we manually inspected a random subset of 100 callback candidates which for which CGMINER did not find an edge. We found that 95% of these callback candidates are indeed not callback methods. 5% were callback candidates that were missed due to not triggering the respective method in an app at runtime, i.e., actual false negatives.

Another approach to check for missing callbacks is to use benchmark suites. To our best knowledge, there is no ground truth benchmark specifically for callback edges. Therefore, we used the artificially generated apps introduced in Section 7.3 as a base. On these apps, CGMINER retrieved edges for 82 % of the callback candidates. For 5%, the generated apps missed at least one method call to actually trigger the callback. For the rest, these are not valid callbacks, i.e. these are true negatives.

7.5 RQ3: Instrumentation Performance

Our implementation of CGMINER integrates into an analysis framework that schedules jobs for processing and performs them when free capacity is available on a system consisting of analysis server, DFarm device farm server, DFarm controllers, etc. We therefore measure the performance of the relevant parts of the analysis individually, because the overall time is dominated by the infrastructure.

First, an APK file is imported into the analysis framework and its code is transformed to Jimple. This step takes 59 seconds on average (minimum: 8s, maximum: 130s, median: 54s). Note that this time also includes decoding the app's resource files and manifest, because

the instrumentation framework assumes that they can be modified as well. In fact, the framework injects an application class (if not yet present), services, and permissions as part of the communication infrastructure between device and analysis server.

After the app has been imported, the instrumentation is performed, which takes 9 seconds on average (minimum: 4s, maximum: 14s, median: 4s). This time includes the part specific to CGMINER, i.e., defining the callback events. The CGMINER part alone never takes more than one second with an average of 0.3 seconds and a mean of 0.2 seconds. Translating the callback event definitions into statements, along with the other required modifications to establish the connection between device and analysis host, is part of the VUSC analysis framework. It counts into the 9 seconds and not the one second. On average, the overall analysis performs 432,000 instrumentation steps (maximum: 556,000 steps, minimum: 302,000 steps, median: 396,000 steps). Each step can be a single statement added or removed, a change to a value in a statement, etc.

Next, the transformed Jimple code and resource files including the manifest are written back into an APK file. This step takes 44 seconds on average (maximum: 52s, minimum: 27s, mean: 43s). The total time spent before running the app is 112 seconds on average (minimum: 40s, maximum: 208s, median: 106s). After building, we run the apps for a fixed period of time and send inputs manually and afterwards by using Monkey. Therefore, measuring the runtime performance is not informative. We observe that the apps still meet the responsiveness requirements of the Android operating system.

We conclude that CGMINER’s runtime is dominated by the dynamic exploration (5 minutes in our configuration), and not by the analysis and instrumentation beforehand (roughly 2-3 minutes). Note that CGMINER is intended to be used as a tool to generate callback summaries as a one-time effort. The performance numbers shown correspond to the time needed to generate the summaries. In contrast, applying the callback summaries generated by CGMINER does not require any dynamic analysis.

7.6 RQ4: Prevalence of Transfer Functions

In contrast to previous approaches from the literature [8], CGMINER supports complex callback registration that require transfer functions. In this research question, we evaluate how important transfer edges are in real-world apps. Conceptually, disregarding transfer functions via approximations may lead to a loss of flow-sensitivity as well as false positive callgraph edges as shown in Section 3.

During callback identification (see Section 7.3), CGMINER discovered CGMINER 2046 edges, 146 of which require transfer functions (6.00%). Note that these numbers are on API level. Even a single transfer edge can be highly important if the respective API is used frequently in apps.

To measure the impact of these 146 edges, we therefore check how often these APIs that require transfer functions are called in real-world apps. To avoid any bias from the apps on which the transfer edges were originally identified, we chose a separate evaluation dataset. We randomly picked 1988 apps from the same Play Store and AndroZoo data source explained in Section 7.1. On this app set, 1928 apps (96.98 %) use transfer functions in their code. On average, each app uses 103.65 different transfer functions. For comparison, apps in the dataset use 1089.24 callbacks on average. These results show that transfer functions are highly relevant in practice.

Table 2 shows the ten most frequently-used transfer functions and their edges together with the number of times the respective transfer function was encountered in our evaluation app set. Six of the most found functions are related to wrapped IO calls. For example, a `read` method call on an `BufferedReader` instance triggers the `read` callback on the reader which was specified during the construction of the `BufferedReader` object.

■ Listing 3 Transfer Function Code.

```

1  StringReader sr = new StringReader(str) {
2      public void close() {
3          // additional callback code
4          super.close();
5      }
6  };
7 BufferedReader br = new BufferedReader(sr);
8 br.read();

```

In other words, the constructor of the `BufferedReader` is a transfer function. Listing 3 shows a code example for such a case. Without modeling the transfer function, approaches such as EdgeMiner must model an edge from the call to the `BufferedReader` constructor in Line 7 to all methods of the `Reader` that is passed as the first argument. In the example, this would even be a call to `close`, even though the `StringReader` is never closed².

In total, 19.55 % of the callbacks that EdgeMiner has identified require transfer functions. All of them are missed. In contrast, CGMINER only misses 3.52% of the transfer functions that are required for the callbacks identified by CGMINER.

7.7 RQ5: Comparison with EdgeMiner

For a comparison on the Android system we used Android 4.2, since the since the Edgeminer paper used Android 4.2 for evaluation. EdgeMiner yields 5,125,472 edges in total for Android 4.2, whereas CGMINER yields 2046 edges. We found that the EdgeMiner output contains reference to non-existing parameters or callbacks with incompatible types. First, we removed these edges automatically. Furthermore, we noticed that EdgeMiner's output may contain multiple callback edges overloads referencing all implementations albeit an edge for the abstract superclass or interface was enough. We therefore removed the edges of these overloads automatically and made sure that the removal process does not change the semantics. After this cleanup, 17298 callback edges remain (0.36% of the original edge set). This constitutes as our new base set for EdgeMiner, which we verified manually.

On this base set we compute a false positive rate of 47.42% for EdgeMiner. Manually checking the CGMINER edges only yields a false positive rate of 1.86%. CGMINER's dynamic analysis avoids the false positives that arise from EdgeMiner's VTA callgraph and the resulting imprecise points-to set for that is used to derive the types of registers / variables that store callback objects. We make available the annotated outputs of EdgeMiner and CGMINER as part of our data package. We removed Android APIs not present in Android 4.2 from CGMINER results, since EdgeMiner cannot possibly have results involving these APIs and apps in RQ1 may use newer API methods than those present in Android 4.2.

Note that EdgeMiner's data is based on the Android system's implementation JAR alone without third-party libraries. For a fair comparison, we used the library detector integrated in VUSC to obtain maven coordinates of libraries used in apps in RQ1. We downloaded the JAR files of the library and executed EdgeMiner on these JARs. Table 1 shows the results for the Android system (comprising the Android SDK and the Java standard library) as well as third party libraries. While EdgeMiner has more edges on

² The `StringReader` has no finalizer either that would call `close`.

the Android system jar, it has significantly more false positives and significantly more incomplete edges than CGMINER. Incomplete edges are edges that lack one or more necessary transfer functions. For example, in the motivating example of Section 3, an edge $\langle \text{HttpTask.cons}, 0 \rangle \rightarrow \langle \text{IHttpCompleted.onCallback}, 1 \rangle$ would be incomplete, because it is missing the necessary transfer edges to `schedule` and `runAll`.

To get a better understanding of the sources of imprecision, we analyzed the false positives and the incomplete edges produced by EdgeMiner in detail. Setters and constructors are particularly relevant sources of imprecision. In total, EdgeMiner reports 2826 constructor edges and 1516 setter edges. EdgeMiner places edges from these methods to the callbacks. In reality, however, these methods do not invoke any callback function, neither directly or transitively. Instead, the references to callback objects are saved into fields. Only later, when other methods are called, these references are read back from the field and the respective callback is invoked.

Table 1 Results on different libraries for CGMINER and EdgeMiner. TP: true positive edges, FP: false positive edges, IE: incomplete edges (missing transfers). Regarding “Other”: We have included several other libraries, which we made sure to supply to EdgeMiner as well.

Library	CGMiner			EdgeMiner		
	TP	FP	IE	TP	FP	IE
Android	1051	27	21	4957	7704	2586
Java	574	8	46	702	494	709
Apache HttpClient	59	0	0	14	1	0
kotlin	46	0	0	0	0	0
Xml Pull Parser	36	0	0	41	4	86
Apache HttpCore	12	0	0	0	0	0
Rxjava	9	0	0	0	0	0
play-services-ads-lite	8	0	0	0	0	0
Gson	7	1	0	0	0	0
Firebase	5	0	0	0	0	0
Google common	4	2	5	0	0	0
play-services-basement	2	0	0	0	0	0
play-services-maps	2	0	0	0	0	0
C3DEngine	1	0	0	0	0	0
AndEngine	1	0	0	0	0	0
Cocos2dx	1	0	0	0	0	0
Other	118	0	0	0	0	0
Total	1936	38	72	5714	8203	3381
Rate	94.62%	1.86%	3.52%	33.03%	47.42%	19.55%

We next describe some examples of such false edges. One constructor of the `ConcurrentSkipListSet` class, for example, takes a `Comparator` as a parameter. A `ConcurrentSkipListSet` is a sorted set, which orders elements according to this comparator. The constructor only saves the comparator instance to a field, and the callback is triggered when a new element is inserted into the set using the `add` or `addAll` methods. EdgeMiner places an edge from the constructor to the Comparator’s `compare` method, although these methods are only called upon adding an element. In total, EdgeMiner reports incomplete edges in 1734 out of the 2826 constructor edges, and 779 are false positives (11.08 % correctness rate). In contrast, CGMINER

yields only 13 incomplete and 16 false positive edges on 323 constructor edges (91.02 %). Similarly, most setters set a field to a specific value and do trigger callbacks. For example, EdgeMiner assumes an edge from `LayoutInflater.setFactory(LayoutInflater$Factory)` to `LayoutInflater$Factory.onCreateView`, although this is only the registration site of the call. Android calls the callback only upon inflating a layout using the `inflate` method. Since EdgeMiner does not support transfer edges, it misses the corresponding transfers on these edges. For EdgeMiner, out of 1516 setter edges, 663 are incomplete and 195 are false positives. This constitutes a correctness rate of 43.4 % on these edges for EdgeMiner. On 467 edges on setter methods reported by CGMINER, 21 are incomplete and 0 false positive, resulting in a correctness rate of 95.5 %.

7.8 RQ6: Case Study on Individual Callbacks

Using CGMINER, we have identified non-obvious multi-step callbacks. The *ActionBarSherlock*³ library allows a developer to integrate a tab view into his app. New tabs are added using `addTab` on an `ActionBar` object which takes the tab as a parameter. With `Tab.setTabListener`, the developer can register a callback that is notified when the user selects the tab. Therefore, `addTab`, which automatically opens the new tab, triggers the `onTabSelected` callback previously registered on the tab. This callback involves two interactive objects, `ActionBar` and `Tab`. Other tools such as EdgeMiner [8] cannot precisely identify and model such a callback. In case of the `AsyncTask`, CGMINER detects that a call to `AsyncTask.execute` results in several callbacks being called: `onPreExecute`, `doInBackground`, `onPostExecute`.

CGMINER identifies similar API calls that trigger multiple callbacks in the API for the SQLite database engine. A call to `getWritableDatabase` or `getReadableDatabase` triggers the callbacks `onOpen`, `onConfigure` and `onCreate`. Some callbacks are triggered by the operating system upon external events, such as new sensor data. In this case, the last user code call site for this callback is the statement that registered the callback. Even though this statement does not immediately invoke the callback, modeling an edge from the registration site to the callback is still a common and useful approximation. CGMINER, for example, finds a connection between Android's `registerListener` method and the `onSensorChanged` of the `SensorEventListener` interface.

7.9 RQ7: Effect on Client Analysis

We next evaluate the effect of callback summaries on data flow analysis. We ran FlowDroid on 200 randomly selected apps, chosen from the same data source already explained in Section 7.1. Note that this data flow analysis is distinct from the data flow analysis we perform in our approach in Section 4.5. The purpose of the data flow analysis in Section 4.5 is to track all container objects that hold callback objects. This is only relevant when generating new summaries. In contrast, this section performs data flow analysis to determine sensitive flows. For this, we use already computed summaries from Section 7.3 to extend the call graph. We configured a timeout of 3 minutes for callgraph construction and 15 minutes for the main data flow analysis. The analysis was assigned 250 GB of heap space and a maximum of 7 cores. This configuration allowed us to parallelize multiple runs on the same machine. We evaluated three different configurations. As our baseline, we perform the FlowDroid data flow

³ <http://actionbarsherlock.com/>

■ **Table 2** The ten most found transfer functions in apps.

Count	Transfer function & Edge
1161	<i>BufferedReader.readLine</i> → ⟨ <i>BufferedReader.cons</i> , -1⟩ → ⟨ <i>InputStreamReader.read</i> , 0⟩
1071	<i>Runnable.run</i> → ⟨ <i>FutureTask.cons</i> , -1⟩ → ⟨ <i>Callable.call</i> , 0⟩
1031	<i>BufferedInputStream.cons</i> → ⟨ <i>GZIPInputStream.cons</i> , 0⟩ → ⟨ <i>AutoCloseable.close</i> , 0⟩
995	<i>InputStream.read</i> → ⟨ <i>BufferedInputStream.cons</i> , -1⟩ → ⟨ <i>FileInputStream.read</i> , 0⟩
983	<i>PrintWriter.print</i> → ⟨ <i>PrintWriter.cons</i> , -1⟩ → ⟨ <i>OutputStreamWriter.write</i> , 0⟩
980	<i>View.layout</i> → ⟨ <i>View.addOnLayoutChangeListener</i> , -1⟩ → ⟨ <i>View\$OnLayoutChangeListener.onLayoutChange</i> , 0⟩
977	<i>Executor.execute</i> → ⟨ <i>ScheduledThreadPoolExecutor.cons</i> , -1⟩ → ⟨ <i>ThreadFactory.newThread</i> , 1⟩
963	<i>OutputStream.write</i> → ⟨ <i>CipherOutputStream.cons</i> , -1⟩ → ⟨ <i>ByteArrayOutputStream.write</i> , 0⟩
945	<i>PrintStream.println</i> → ⟨ <i>PrintWriter.cons</i> , -1⟩ → ⟨ <i>FileWriter.write</i> , 0⟩
920	<i>Parcel.writeBundle</i> → ⟨ <i>Parcel.writeStrongBinder</i> , -1⟩ → ⟨ <i>ffm.dispatchTransaction</i> , 0⟩

analysis without any callback edges. We then ran FlowDroid again with callback summaries generated by EdgeMiner and with summaries generated by CGMINER. For each run, we recorded the discovered flows.

■ **Listing 4** Callback Parameter Analysis.

```

1 class MyTaskRunnable implements Runnable {
2     public String data;
3     public void run() {
4         sink(data);
5     }
6 }
7 ThreadPoolExecutor executor = new ThreadPoolExecutor(...);
8 Runnable r = new MyTaskRunnable();
9 r.data = source();
10 executor.execute(r);

```

Recall that CGMINER and EdgeMiner only model control flow, but not data flow. In the example in Listing 4, the first parameter of Line 10 becomes the base object inside the callee `run`. This relationship is important, because the field `data` inside the callback object, i.e., the access path `r.data`, is tainted in Line 9. When the data flow analysis processes the sink call in Line 4, the taint must be available as `this.data`. In other words, FlowDroid's IFDS call edge must re-write the access path from `r.data` to `this.data`.

Neither CGMINER nor EdgeMiner create data flow summaries. Therefore, our initial runs had the required callgraph edges, but could not track data flows across the callback edges. With this configuration, our baseline yielded 2021 flows. With EdgeMiner summaries, FlowDroid found 3575 flows (77 % more than baseline). With CGMINER summaries, 2554 flows were detected, which is 26 % more than the baseline. As expected, FlowDroid discovers

more flows when provided with callback summaries. Further, since EdgeMiner has more (true positive) callback edges than CGMINER as shown in Table 1, it is unsurprising that EdgeMiner leads to more flows as well. FlowDroid tracks flows across the interprocedural data flow graph. Every additional callgraph edge has the potential to lead to more flows.

We next augment the callgraph summaries with data flow information using heuristics. Firstly, we map the base object on which the callback is invoked (which is known from the callback summary) to the `this` object of the callee. In the example in Listing 4, this leads to a data flow edge from variable `r` to the `this` object of the callback. This allows FlowDroid to map `r.data` to `this.data`. Secondly, if there is an edge from a call site to a callback method and the call site accepts a parameter that is cast-compatible to a parameter of the callback method, we assume a data flow edge. We stress that these heuristics are not meant to be complete. We use them as part of our evaluation to better estimate the effect of callback edges for data flow analyses.

With these data flow mappings, FlowDroid finds 2717 flows with EdgeMiner and 2830 flows (40 % more than the baseline) with CGMINER. In the baseline without callback summaries, no data flow mapping is possible. We observe that when we use parameter mappings, FlowDroid with CGMINER finds more data flows than FlowDroid with EdgeMiner, although CGMINER has vastly fewer callgraph summary edges than EdgeMiner. The number of flows found using EdgeMiner callbacks drops from 3575 flows when using no parameter mapping to 2717 when using parameter mappings. The explanation lies in FlowDroid’s sanity checking. For example, when FlowDroid propagates taints along edges in the interprocedural control flow graph, it propagates types along as well. In each propagation step, these propagated types are checked for cast-compatibility with the target variables. EdgeMiner’s spurious callback edges lead to many cast-incompatible propagations. This leads to taints being discarded. For EdgeMiner with many false positive edges or incomplete edges, i.e., edges placed at the wrong statement, this leads to a significant amount of flows being discarded. On the other hand, the increase in data flows with CGMINER summaries represents actual taint propagation along the callback edges. This is expected for examples such as the one shown in Listing 4. Since CGMINER only has few false positives, it is almost unaffected by FlowDroid’s type checks, but benefits from parameter mappings being available. We then analyzed the correctness of the data flows. Due to the large amount of data flow results, we only looked at a subset of 50 flows of each evaluation run. EdgeMiner shows a precision of 94.34% on these flows. Recall that FlowDroid already discards flows with cast-incompatible assignments along the taint propagation path. Therefore, it can eliminate some false-positive flows during propagation. CGMINER delivers a true positive rate of 100%.

As stated in the beginning of this section, we evaluated on FlowDroid using 200 randomly selected apps. From these apps, we found that 94 % invoke at least one callback method. Filtering apps with no callback methods yields the following results: Without data flow mappings, the baseline has 1,987 flows (compared to 2,021 flows w/o filtering). With EdgeMiner summaries, FlowDroid finds 3,505 flows (compared to 3,575). With CGMINER summaries, FlowDroid finds 2,511 flows (compared to 2,554). With data flow mappings and EdgeMiner summaries, FlowDroid then finds 2,681 flows (compared to 2,717). With data flow mappings and CGMINER summaries, FlowDroid finds 2,787 flows (compared to 2,830).

8 Related Work

EdgeMiner [8] statically analyzes the Android framework to build models for callbacks in API methods. Due to the large code size of the Android framework, EdgeMiner over-approximates virtual dispatch using a CHA callgraph. It further suffers from the inherent challenges

of static analysis, such as dealing with reflective method calls. CGMINER avoids such imprecision and only generates edges that are possible at runtime. EdgeMiner tries to find registration and callback pairs using def-use chains. The search starts at a potential callback and follows the definitions of the base object through the library code until it reaches the start of a method that has no more potential callers within the library. In case the callback object is read from a field on the path, all writes to the field are considered as potential definitions and are thus followed, regardless of their context. EdgeMiner does not provide support for collections or arrays and would not be able to generate a correct summary for our example from Listing 3. Perez and Le [20] present Predicate Callback Summaries (PCS) that model under which conditions a callback or Android lifecycle method is invoked. Their static tool Lithium works on the Android source code and suffers from the same challenges of large-scale static callgraph analysis as EdgeMiner. It does not support our complex example either. Callback Control Flow Automata (CCFA) [21] integrates PCS and Window Transition Graphs (WTGs) [27], and focuses mainly on UI callbacks and lifecycle methods. We consider integrating a predicate analysis into CGMINER as future work. Zhang and Ryder [31] propose a static library analysis based on data reachability. Similarly, Guo et al. present an approach based on backward data dependency analysis [12]. These analyses must be conducted for each call site, which is costly in practice [15].

Some work has focused on Android UI callbacks [28], e.g., for context-sensitive linking of parameterized callbacks to their respective UI elements. The same callback may be used for multiple buttons. The clicked button is passed to the callback as a parameter, and the shared implementation may follow different control flow paths depending on that parameter value. The information which API methods may trigger callbacks is usually an external input to these algorithms, which CGMINER can provide. Other work has increased the coverage of dynamic analyses by reasoning about UI callbacks using a combination of static and dynamic analysis [6, 29]. CGMINER is more generic and therefore cannot exploit specific properties of Activities or Intents. TamiFlex [7] uses dynamic analysis to record the runtime values at reflective method calls and build models of known callees for such call sites. Harvester [22] uses static slicing and dynamic execution to extract runtime values at reflective call sites. It rewrites these call sites into explicit calls to deobfuscate apps. The outputs of these approaches are specific to a concrete target program and do not generalize over re-usable libraries. HeapDL [11] uses heap dumps to reconstruct callgraph edges. It can be used to discover callback registration methods, which directly or transitively call callback methods when such a call is present on the stack of some thread in the heap dump. However, when a callback is saved as a field during callback registration and used later on, this approach would require multiple heap dumps taken at precisely the correct timings. Otherwise, either the callback registration, the callback invocation or both are missed. StubDroid [2] statically generates data flow summaries for libraries. It requires a complete and precise callgraph of the library to work properly and can therefore benefit from the callgraph models generated by CGMINER. Our callback summaries are relevant for various analyses (power analysis, privacy analysis, injection analysis, etc.) that currently rely on manual callback models [5, 30, 18, 13].

9 Conclusion

We have presented CGMINER, an approach for dynamically monitoring apps to derive callback summaries for commonly-used libraries. These summaries can then be applied to static analyses that require a callgraph. We have shown that CGMINER yields a precision of more than 94%. With the CGMINER summaries, FlowDroid detects 40 % more flows in comparison to our baseline. In the future, we will run CGMINER on more apps to generate and provide to the community summaries of lesser-used libraries.

Data Availability. The data and implementation have been published to <https://github.com/Fraunhofer-SIT/DynamicCallbackSummaries/>. Since CGMINER is built upon the VUSC commercial scanner, you need to apply for a free academic license for VUSC to build and run CGMINER.

References

- 1 Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 468–471. IEEE, 2016.
- 2 Steven Arzt and Eric Bodden. Stubdroid: automatic inference of precise data-flow summaries for the android framework. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 725–735. IEEE, 2016.
- 3 Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Instrumenting android and java applications as easy as abc. In *International Conference on Runtime Verification*, pages 364–381. Springer, 2013.
- 4 Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Susi: A tool for the fully automated classification and categorization of android sources and sinks. *University of Darmstadt, Tech. Rep. TUDCS-2013-0114*, 2013.
- 5 Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.
- 6 Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 641–660, 2013.
- 7 Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 241–250. IEEE, 2011.
- 8 Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. Edgeminer: Automatically detecting implicit control flow transitions through the android framework. In *NDSS*, 2015.
- 9 Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: Are we there yet? (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 429–440, 2015. doi:10.1109/ASE.2015.89.
- 10 Google, Inc. Ui/application exerciser monkey, 2023. URL: <https://developer.android.com/studio/test/other-testing-tools/monkey>.
- 11 Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. Heaps don't lie: countering unsoundness with heap snapshots. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–27, 2017.
- 12 Chenkai Guo, Quanqi Ye, Naipeng Dong, Guangdong Bai, Jin Song Dong, and Jing Xu. Automatic construction of callback model for android application. In *2016 21st International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 231–234. IEEE, 2016.
- 13 Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L Pereira, Gilles A Pokam, Peter M Chen, and Jason Flinn. Race detection for event-driven mobile applications. *ACM SIGPLAN Notices*, 49(6):326–336, 2014.
- 14 Patrick Lam, Eric Bodden, Ondrej Lhotak, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*, oktober 2011.

- 15 Ondrej Lhoták. Comparing call graphs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 37–42, 2007.
- 16 Ondřej Lhoták and Laurie Hendren. Scaling java points-to analysis using spark. In Görel Hedin, editor, *Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 153–169. Springer Berlin Heidelberg, 2003. doi:[10.1007/3-540-36579-6_12](https://doi.org/10.1007/3-540-36579-6_12).
- 17 Li Li, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. An investigation into the use of common libraries in android apps. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 403–414, 2016. doi:[10.1109/SANER.2016.52](https://doi.org/10.1109/SANER.2016.52).
- 18 Yepang Liu, Chang Xu, and Shing-Chi Cheung. Where has my battery gone? finding sensor related energy black holes in smartphone applications. In *2013 IEEE international conference on pervasive Computing and Communications (PerCom)*, pages 2–10. IEEE, 2013.
- 19 Marc Miltenberger, Julien Gerdig, Jens Guthmann, and Steven Arzt. Dfarm: massive-scaling dynamic android app analysis on real hardware. In *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*, pages 12–15, 2020.
- 20 Danilo Dominguez Perez and Wei Le. Generating predicate callback summaries for the android framework. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 68–78. IEEE, 2017.
- 21 Danilo Dominguez Perez and Wei Le. Specifying callback control flow of mobile apps using finite automata. *IEEE Transactions on Software Engineering*, 47(2):379–392, 2021. doi:[10.1109/TSE.2019.2893207](https://doi.org/10.1109/TSE.2019.2893207).
- 22 Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. In *NDSS*, 2016.
- 23 Siegfried Rasthofer, Steven Arzt, Stefan Triller, and Michael Pradel. Making malory behave maliciously: Targeted fuzzing of android execution environments. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 300–311. IEEE, 2017.
- 24 Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '00*, pages 264–280, New York, NY, USA, 2000. Association for Computing Machinery. doi:[10.1145/353171.353189](https://doi.org/10.1145/353171.353189).
- 25 Raja Vallee-Rai and Laurie J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations, 1998.
- 26 Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of google play. In *The 2014 ACM international conference on Measurement and modeling of computer systems*, pages 221–233, 2014.
- 27 Shengqian Yang, Haowei Wu, Hailong Zhang, Yan Wang, Chandrasekar Swaminathan, Dacong Yan, and Atanas Rountev. Static window transition graphs for android. *Automated Software Engineering*, 25(4):833–873, 2018.
- 28 Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static control-flow analysis of user-driven callbacks in android applications. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 89–99. IEEE, 2015.
- 29 Wei Yang, Mukul R Prasad, and Tao Xie. A grey-box approach for automated gui-model generation of mobile applications. In *International Conference on Fundamental Approaches to Software Engineering*, pages 250–265. Springer, 2013.
- 30 Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X Sean Wang. Appintent: Analyzing sensitive data transmission in android for privacy leakage detection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1043–1054, 2013.
- 31 Weilei Zhang and Barbara G Ryder. Automatic construction of accurate application call graph with library call abstraction for java. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(4):231–252, 2007.

Behavioural Up/down Casting For Statically Typed Languages

Lorenzo Bacchiani  

University of Bologna, Italy

Mario Bravetti  

University of Bologna, Italy

Marco Giunti  

University of Oxford, UK

João Mota  

NOVA LINCS, Nova University Lisbon, Portugal

NOVA School of Science and Technology, Caparica, Portugal

António Ravara  

NOVA LINCS, Nova University Lisbon, Portugal

NOVA School of Science and Technology, Caparica, Portugal

Abstract

We provide support for polymorphism in static typestate analysis for object-oriented languages with upcasts and downcasts. Recent work has shown how typestate analysis can be embedded in the development of Java programs to obtain safer behaviour at runtime, e.g., absence of null pointer errors and protocol completion. In that approach, inheritance is supported at the price of limiting casts in source code, thus only allowing those at the beginning of the protocol, i.e., immediately after objects creation, or at the end, and in turn seriously affecting the applicability of the analysis.

In this paper, we provide a solution to this open problem in typestate analysis by introducing a theory based on a richer data structure, named typestate tree, which supports upcast and downcast operations at any point of the protocol by leveraging union and intersection types. The soundness of the typestate tree-based approach has been mechanised in Coq.

The theory can be applied to most object-oriented languages statically analysable through typestates, thus opening new scenarios for acceptance of programs exploiting inheritance and casting. To defend this thesis, we show an application of the theory, by embedding the typestate tree mechanism in a Java-like object-oriented language, and proving its soundness.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Object oriented constructs; Theory of computation → Program verification

Keywords and phrases Behavioural types, object-oriented programming, subtyping, cast, typestates

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.5

Supplementary Material Software (Coq Proofs Artifact): <https://zenodo.org/records/7712822>

Software (JaTyC Tool Artifact): <https://zenodo.org/records/7712915>

Software (JaTyC Tool on GitHub): <https://github.com/jdmota/java-typestate-checker>

archived at [swh:1:dir:69edd64b73a190021dd96ee97c7192722edfd00f](https://scholar.archive.org/2024/05/29/64b73a190021dd96ee97c7192722edfd00f)

Funding This work was partially supported by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant agreement No. 778233 (BehAPI).

Marco Giunti: EPSRC (EP/T006544/2).

João Mota: NOVA LINCS (UIDB/04516/2020) via the Portuguese Fundação para a Ciência e a Tecnologia (doi:10.54499/2021.05297.BD).

 © Lorenzo Bacchiani, Mario Bravetti, Marco Giunti, João Mota, and António Ravara;
licensed under Creative Commons License CC-BY 4.0

38th European Conference on Object-Oriented Programming (ECOOP 2024).

Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 5; pp. 5:1–5:28

 Leibniz International Proceedings in Informatics

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

1 Introduction

Modern software engineering practices, e.g., Continuous Delivery [21], produce reliable software at high pace, through automatic pipelines of building, testing, etc. However, programming errors such as dereferencing null pointers [19] or using objects wrongly (e.g. reading from a closed file; closing a socket that timed out¹) are often subtle and difficult to catch, even during the automated testing process. As put by Dijkstra [14]: “*program testing can be used to show the presence of bugs, but never to show their absence*”. So, tools to (statically) catch bugs are essential. Formal methods like deductive verification are difficult to adopt given the effort required [26], but lightweight static program analysis techniques can greatly improve the quality of the source code by detecting at compile-time logic errors, i.e., an unexpected action or behaviour. Beckman et al. [6] observe:

In the open-source projects in our study [...] approximately 7.2% of all types defined protocols, while 13% of classes were clients of types defining protocols. [...] This suggests that protocol checking tools are widely applicable.

To tackle the challenge of finding bugs in object-oriented code, where objects naturally have protocols, in this paper we provide a protocol checking approach, supported by a tool, based on typestates [31, 15]. The work we present is applicable to most object-oriented languages, following the approach in closely related work [18, 11]: attach protocols (essentially, allowed orders of method calls) to classes and type check classes (i.e., their method bodies) following the protocol, thus gaining typestate-based nullness checking (ensuring memory-safety), protocol compliance, and protocol completion (under program termination).

In our previous work [3], we applied the approach to Java, proposing the JaTyC tool, exploiting the seminal simulation-based notion of subtyping [16] to check that the protocol of a class was a subtype of the protocol of its superclass. However, only upcasts and downcasts at the beginning of an object protocol (i.e., just after object creation) or at the end (i.e., in the `end` state) were allowed. Additionally, to determine if a typestate was a subtype of another, the simulation was only applied to the initial typestates of the protocols. It is crucial to overcome these limitations to make JaTyC applicable to real-world scenarios since, as shown in the study of Mastrangelo et al. [25], *casts are widely used*. The type checker was developed following a research methodology based on an iterative/incremental approach (see figure in **Appendix A** for details), based on the theory, which together with motivating examples, drove the type checker implementation (built upon the Checker Framework [30]).

Running example. To emphasise the relevance of our contribution, consider an example inspired from the automotive sector where driving dynamics control allows to customise the drive mode²; for SUVs, in particular, we consider a `Comfort` and a `Sport` modalities, where each allows specific features: `EcoDrive` and `FourWheelsDrive`, respectively.³ List. 1 and List. 2 describe the behaviours of the controllers of a `Car` and a `SUV`, respectively, where class `SUV` extends `Car`. All cars have two base states: `OFF`, which models a powered off car, and `ON`, which represents a powered on car that can perform certain actions, e.g., set a concrete speed. In `OFF`, it is possible to `turnOn` the car and then access features like `setSpeed`. Dually, in `ON`, it is possible to `turnOff` the car. The `turnOn` action may, by some

¹ <https://github.com/redis/jedis/issues/1747>.

² BMW Sport vs Comfort modes: bmwofstratham.com/bmw-sport-mode-vs-comfort-mode-stratham-nh

³ Code online: github.com/jdmota/java-typestate-checker/tree/master/examples/car-example

Listing 1 Car protocol.

```

1 typestate Car {
2   OFF = {
3     boolean turnOn(): <true:ON, false:OFF>,
4     drop: end
5   }
6   ON = {
7     void turnOff(): OFF,
8     void setSpeed(int): ON
9   }
10 }
11 }
```

Listing 2 SUV protocol (SUV extends Car).

```

1 typestate SUV {
2   OFF = {
3     boolean turnOn(): <true:COMF_ON, false:OFF>,
4     drop: end
5   }
6   COMF_ON = {
7     void turnOff(): OFF,
8     void setSpeed(int): COMF_ON,
9     Mode switchMode(): <SPORT:SPORT_ON, COMFORT:COMF_ON>,
10    void setEcoDrive(boolean): COMF_ON
11  }
12   SPORT_ON = {
13     void turnOff(): OFF,
14     void setSpeed(int): SPORT_ON,
15     Mode switchMode(): <SPORT:SPORT_ON, COMFORT:COMF_ON>,
16     void setFourWheels(boolean): SPORT_ON
17  }
18 }
```

technical reason, fail, and so, depending on the returned value, either the resulting case is `ON` or `OFF`. SUVs are described by the protocol in Listing 2: when they are successfully powered on by means of `turnOn`, they are set in Comfort mode (`COMF_ON`), and in turn they enjoy specific operations, e.g., `setEcoDrive`. The mode can be changed by executing `switchMode`, whose result depends on the reached mode being still Comfort (as the operation may fail, e.g., if the speed is too high), or Sport (`SPORT_ON`). Similarly, the Sport mode provides the `switchMode` actions and also specific ones, e.g., `setFourWheels`. Note that `setSpeed` is overridden in the SUV class: if eco-drive is active, the speed must respect a given threshold, otherwise it can be set to any value. As we will see, in Section 6, overriding correctness is checked based on typestate variance, thus dynamic dispatch is guaranteed to work safely. Section 8 explains how our work compares with others dealing with inheritance.

Each protocol is defined by a set of *typestates* (e.g., in List. 1, `OFF` and `ON`), each one defining a set of callable methods and subsequent states, possibly depending on return values: e.g., if `turnOn` returns `true` in state `OFF` of the SUV protocol, then the next state is `COMF_ON`. By applying the subtyping algorithm by Gay and Hole [17] to the initial typestates (i.e., `OFF` in Car and SUV protocols), we see that the SUV protocol is a subtype of the Car one.

Listing 3 upcast/downcast limitation protocol.

```

1 public static void dispatch(@Requires("ON") Car c) { ... }
2 public static void providePoweredSUV(@Requires("OFF") SUV c) {
3   if (c.turnOn()) dispatch(c); // Upcast rejected by current typestate analysis
4 }
```

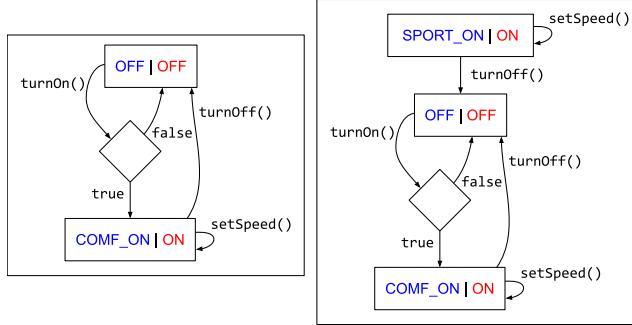
Key insight. Even for simple classes as `Car` and `SUV`, limiting casts only at the beginning/end of the protocols seriously reduces the programs we are able to typestate-check, such as the one in List. 3, where an automotive system dispatches already powered on cars (i.e., required in typestate `ON`), whether they are SUVs or not. Removing this limitation is challenging. The solution relies on the key insight that one has to run the subtyping algorithm not only on the pair of initial typestates, but on all pairs, to find all typestates in both protocols that are in a subtyping relation. For example, the `limitSpeed` method in List. 4 expects a `Car` in typestate `ON`. Since `SPORT_ON` is a subtype of `ON`, code passing a `SUV` in typestate `SPORT_ON` to `limitSpeed` is type-safe. However, if we run the subtyping algorithm starting from the pair of initial typestates of the given protocols (i.e., `(OFF, OFF)`), the generated simulation relation [4, 17] (in Figure 1, where boxes represent input states, and diamonds output ones), will not include `(SPORT_ON, ON)` (leftmost graph). If we provide `(SPORT_ON, ON)`, we realise that this pair is in the *typestate subtyping* relation (rightmost graph).

■ Listing 4 Limitation of the Subtyping Algorithm Application.

```

1 void limitSpeed(@Requires("ON") Car c, int speed) {
2     c.setSpeed(Math.min(speed, 50));
3 }

```



■ Figure 1 Subtyping simulations starting with (OFF, OFF) (left) and (SPORT_ON, ON) (right). Blue depicts typestates of the SUV protocol and red those of Car.

A theory of typestate upcast and downcast. With this key insight, we devise the following mechanism: when downcasting, we look for the typestates (in the protocol of the target class) that are subtypes of the current one; when upcasting, we look for the typestates that are supertypes of the current one. Since multiple typestates may be found, we need a structured notion of *types* to combine them. When downcasting, we combine the subtypes in a *union type* [5, 29] (modelling the fact that the actual type is unknown) so that a method call is allowed only if it is permitted by both elements of the union. Union types are also useful to allow branching code to be typed with different types so that subsequent code, complying with either branch, is accepted (e.g., after an `if` statement). This is more flexible than some other approaches (like session types ones [32]), which require both branches to have the same type. When upcasting, dually, we combine the supertypes in an *intersection type* so that a method call is allowed if it is permitted by at least one element of the intersection.

However, we need more than just intersection and union types. To illustrate the problem, consider a class for an Electric Car (`ECar`) which also extends `Car`. Consider the code in List. 5. After the `if` statement, is `c` an instance of `SUV` or `ECar`? Because of these scenarios, we associate classes with *types* and track all possibilities. To that end, we introduce *typestate trees*, which resemble the class hierarchy. Herein, the typestate tree would have a root for class `Car`, with child nodes for `SUV` and `ECar`. Each node corresponds to a class and maps to the type of the object, accounting for the case in which the object is indeed an instance of that class. In this way, in case of a future downcast to the `SUV` or `ECar` classes, we just consider the corresponding subtree corresponding to that class.

■ Listing 5 Typestate Tree motivation.

```

1 Car c;
2 if (cond) c = new SUV();
3 else c = new ECar();

```

Discussion. The solution we devise is language agnostic, applicable to many object-oriented languages. To test its expressiveness, we applied it to Java, extending JaTyC to now support (up/down)casting in the middle of a protocol. By doing this, we advance related work (Section 8). Kouzapas et al. [24] mention, in the future work section, that to cope

with protocol inheritance between two classes one just needs “a subtyping relation between their typestate specifications”. This is enough if one is only concerned with extending class inheritance but, as we showed, is not adequate to deal with casting, a very common feature.

Furthermore, we support *droppable typestates* (see typestate OFF in List. 1), the final typestates of a protocol – where one can either safely stop using the protocol or perform more actions (if there are any). A droppable typestate with no actions is similar to the **end** state in session types. To fully support droppable typestates, we provide a definition of subtyping over these, extending Gay and Hole’s session type subtyping definition.

Contributions. In short, the main contributions of this work are:

- sound support for safely performing **upcasts** and **downcasts** at any point of a protocol (assuming class downcasts are performed to a class of which an object is a subtype of);
- formalisation of **subtyping over droppable typestates** (generalising Gay and Hole’s session type subtyping);
- **mechanisation** of all definitions and proofs of our results in Coq (artifact available);
- implementation of the presented concepts in our **type checker for Java**, **JaTyC**, where we successfully run all examples included in this paper.

Advances with respect to the state of the art. As far as we know, no previous work deals with casts in the middle of protocols. Moreover, droppable states allow to mark states when the protocol can be safely stopped, another key concept. So, our work advances the state of the art with expressive support for inheritance and casting in object-oriented languages, leveraging on behavioural types [2, 22].

Structure of the paper. **Section 2** presents the subtyping relation, shown to be a pre-order, and a complete and sound algorithm to check typestates subtyping (Theorems 8 and 9). **Section 3** presents upcast and downcast, and the crucial result that each operation preserves subtyping (Theorems 23 and 28). Moreover, we show that, as expected, each operation reverses the other (Corollaries 29 and 30). Finally, we show that method calls make the typestates evolve, preserving subtyping (Theorem 33), and evolution on types commutes with upcast and downcast (Theorems 34 and 35). **Section 4** presents typestate trees, the crucial structure to allow up/down-casting in the middle of a protocol, and main results (Theorem 46 and 51). **Section 5** presents a key result to safely equip a programming language with our subtypestate mechanism – operations on typestate trees preserve their soundness (Theorem 54). **Section 6** discusses how to safely develop a type checking system with typestate trees. Notice that the main contribution of this paper is the provably safe subtypestates theory, paving the way to its use in (most) object-oriented languages. **Section 7** explains how the example presented in List. 6 is type checked with our tool and describes a suite of examples showing the expressiveness of our approach. **Section 8** discusses related work. **Section 9** concludes by envisioning future challenges, e.g., the mechanisation of a type(state tree)-safe object calculus with inheritance to use as basis for our Java implementation. **Appendix A** provides insights on the research methodology we adopted, while **Appendix B** provides a glossary listing all the notations used in the paper.

2 Types and subtyping

In this section, we present the types one can assign to terms of an object-oriented language taken into account, and the corresponding subtyping relations. We first describe typestates, which encode the current state of an object and specify the available methods (Section 2.1).

Then, we compose these in union and intersection types (Section 2.2). Unions track the possible typestates an object might be in, and intersections combine behaviour of two distinct typestates. These will be important when we present how casting works (Section 3).

2.1 Typestates

The following grammar (Def. 1) defines our typestates language. It is very similar to the one presented by Bravetti et al. [11]. The meta-variable m ranges over the set of *method identifiers* **MNames**, o ranges over the set of *output values* **ONames**, and s ranges over the set of *typestate names* **SNames**. The wide tilde stands for a sequence of values.

States are basically of two forms: input and output states. *Input states* $d\{\tilde{m : w}\}$ denote a set like $\{m_1:w_1, m_2:w_2, \dots, m_n:w_n\}$ offering methods (being $n \geq 0$ a natural number), seen as input actions (i.e., external choices), followed by arbitrary states; the meaning is that by selecting method m_i , the input state transitions to state w_i . Input states may optionally be marked as *droppable* (with the subscript `drop` at the left of the set). This marks which input states are final. For example, in List. 1, the typestate **OFF** is defined as a droppable input state (which in the user defined protocol associated with the Java code is represented by the `drop:end` option). *Output states* $\langle \tilde{o : u} \rangle$ denote a set like $\langle o_1:u_1, o_2:u_2, \dots, o_n:u_n \rangle$, presenting all possible outcomes of a method call (values o_1 to o_n , being n a positive natural number), seen as output actions (i.e., internal choices), followed by input states or typestate names. We only consider boolean and enumeration values as outputs.

To deal with recursive behaviour, protocols use equational definitions of typestates.

► **Definition 1.** Typestates, ranged over by meta-variable u , are terms generated by the following grammar. States are terms ranged over by meta-variable w .

$$\begin{array}{lll} u ::= d\{\tilde{m : w}\} \mid s & d ::= \varepsilon \mid \text{drop} \\ w ::= u \mid \langle \tilde{o : u} \rangle & E ::= s = d\{\tilde{m : w}\} \end{array}$$

We assume that in $\langle \tilde{o : u} \rangle$ we have at least one output, while in $d\{\tilde{m : w}\}$, we can have no inputs: `drop{}` represents the protocol ending state, also denoted by `end`. Moreover, in an equation E , typestate names s do not occur unguarded (i.e., we disregard equations like $s = s'$). We write $w^{\tilde{E}}$ to denote state w associated with a set of defining equations. Therefore, we assume that each typestate name s that is used in w and in the body of equations \tilde{E} has a unique defining equation in \tilde{E} . Let \mathcal{W} be the set of terms $w^{\tilde{E}}$ and \mathcal{U} be the set of terms $u^{\tilde{E}}$. Furthermore, let \mathcal{X} be the subset of \mathcal{W} containing only input states $d\{\tilde{m : w}\}$ and \mathcal{Y} be the subset of \mathcal{W} with only output states $\langle \tilde{o : u} \rangle$. Meta-variables x and y range over \mathcal{X} and \mathcal{Y} , respectively. Hereafter, whenever the finite set of equations \tilde{E} is clear from the context, we consider states w implicitly associated with \tilde{E} . Moreover, we omit writing ε .

We can use the grammar introduced in Def. 1 to formally specify *protocols* associated to classes. A protocol is represented by $s^{\tilde{E}}$, with s being the initial typestate name. For example, the protocol associated with class **Car** (List. 1) is **OFF** $^{E_{\text{car}}}$ with E_{car} being:

$$\begin{array}{ll} \text{OFF} & = \text{drop}\{ \text{turnOn} : \langle \text{true : ON}, \text{false : OFF} \rangle \} \\ \text{ON} & = \{ \text{turnOff : OFF}, \text{setSpeed : ON} \} \end{array}$$

The **OFF** typestate is marked as *droppable* and offers a single method (i.e., `turnOn`) which, depending on the returned value (`true` or `false`), leads to either **ON** or **OFF**, respectively. The **ON** typestate offers two methods, `turnOff` and `setSpeed`, leading to **OFF** and **ON**, respectively.

State subtyping is key to support behavioural casting. In our setting, subtypes offer a superset of the supertype methods (input contravariance), and a subset of the supertype outputs (output covariance). To define it properly (with the intended properties), we follow

the work by Gay and Hole on session types subtyping [17]. Therefore, we define the subtyping relation as a simulation one (as protocols can be infinite state systems), and present a sound and complete algorithm to check if one state is a subtype of another. We first introduce function \mathbf{def} to unfold typestate name definitions. The simulation relation follows.

► **Definition 2** (Typestate name definitions). *Function $\mathbf{def} : \mathcal{W} \rightarrow \mathcal{W} \setminus \mathbf{SNames}$ is such that, given a state $w^{\tilde{E}} \in \mathcal{W}$, if it is a typestate name, $\mathbf{def}(w^{\tilde{E}})$ yields the body of its defining equation; otherwise, $\mathbf{def}(w^{\tilde{E}})$ yields the given state $w^{\tilde{E}}$. Formally,*

$$\mathbf{def}(w^{\tilde{E}}) = \begin{cases} x^{\tilde{E}} & \text{if } w^{\tilde{E}} = s^{\tilde{E}' \cup \{s=x\}} \text{ for some } s, E', x \\ w^{\tilde{E}} & \text{otherwise} \end{cases}$$

► **Definition 3** (State simulation). *A relation $R \subseteq \mathcal{W} \times \mathcal{W}$ is a state simulation, if $(w_1^{E_1}, w_2^{E_2}) \in R$ implies the following conditions:*

1. *If $\mathbf{def}(w_1^{E_1}) = d_1\{\widetilde{m:w}\}_1^{E_1}$ then $\mathbf{def}(w_2^{E_2}) = d_2\{\widetilde{m:w}\}_2^{E_2}$ and:*
 - a. *for each $m':w'_2$ in $\{\widetilde{m:w}\}_2$, there is w'_1 such that $m':w'_1$ in $\{\widetilde{m:w}\}_1$ and $(w'_1^{E_1}, w'_2^{E_2}) \in R$.*
 - b. *if $d_2 = \mathbf{drop}$ then $d_1 = \mathbf{drop}$.*
2. *If $\mathbf{def}(w_1^{E_1}) = \langle \widetilde{o:u} \rangle_1^{E_1}$ then $\mathbf{def}(w_2^{E_2}) = \langle \widetilde{o:u} \rangle_2^{E_2}$ and:*
 - a. *for each $o':u_1$ in $\langle \widetilde{o:u} \rangle_1$, there is u_2 such that $o':u_2$ in $\langle \widetilde{o:u} \rangle_2$ and $(u_1^{E_1}, u_2^{E_2}) \in R$.*

Now we define subtyping, following standard approaches.

► **Definition 4** (Subtyping on states). *We say w_1 is a subtype of w_2 , i.e., $w_1^{E_1} \leq_S w_2^{E_2}$, if and only if there exists a state simulation R such that $(w_1^{E_1}, w_2^{E_2}) \in R$.*

An example of a state simulation (Def. 3) follows (also depicted in rightmost graph of Figure 1). It is then easy to check, using Definition 4, that $\mathbf{SPORT_ON}^{E_{SUV}} \leq_S \mathbf{ON}^{E_{Car}}$.

$$\{(\mathbf{SPORT_ON}^{E_{SUV}}, \mathbf{ON}^{E_{Car}}), (\mathbf{OFF}^{E_{SUV}}, \mathbf{OFF}^{E_{Car}}), \\ (\langle \mathbf{true} : \mathbf{COMF_ON}, \mathbf{false} : \mathbf{OFF} \rangle^{E_{SUV}}, \langle \mathbf{true} : \mathbf{ON}, \mathbf{false} : \mathbf{OFF} \rangle^{E_{Car}}), \\ (\mathbf{COMF_ON}^{E_{SUV}}, \mathbf{ON}^{E_{Car}})\}$$

Notice that the common rule for session type subtyping of `end` states (i.e., `end` \leq_S `end`) is derivable from the previous definitions by just picking the relation $R = \{(\mathbf{drop}\{\}, \mathbf{drop}\{\})\}$ and observing that it is a state simulation (Def. 3), thus $\mathbf{drop}\{\} \leq_S \mathbf{drop}\{\}$ holds by Def. 4.

As a sanity check, we show basic subtyping properties on states: reflexivity and transitivity.

► **Lemma 5** (Reflexivity). *For all w , then $w \leq_S w$.*

► **Lemma 6** (Transitivity). *For all $w_1^{E_1}, w_2^{E_2}, w_3^{E_3}$, if $w_1^{E_1} \leq_S w_2^{E_2}$ and $w_2^{E_2} \leq_S w_3^{E_3}$, then also $w_1^{E_1} \leq_S w_3^{E_3}$.*

Defining an algorithm to check state subtyping is crucial, not only because it shows that subtyping is decidable, but also for implementing a type checking procedure (Def. 7). To obtain an algorithm for checking state subtyping, we guarantee termination by always applying ASSUMP, whenever applicable. The initial goal of the algorithm is the judgement $\emptyset \vdash w_1^{E_1} \leq_{S_{alg}} w_2^{E_2}$. This approach is similar to the session type subtyping algorithm presented by Gay and Hole [17]. We also show in Theorems 8 and 9 that the subtyping algorithm is complete and sound with respect to the coinductive definition \leq_S (Def. 4).

► **Definition 7** (Algorithmic state subtyping). *The following inference rules define judgements $\Sigma \vdash w_1^{E_1} \leq_{S_{alg}} w_2^{E_2}$ in which Σ is a set of typestate pairs, containing assumed instances of the subtyping relation.*

$$\frac{(w_1^{E_1}, w_2^{E_2}) \in \Sigma}{\Sigma \vdash w_1^{E_1} \leq_{S_{alg}} w_2^{E_2}} \text{ASSUMP}$$

$$\frac{\begin{array}{c} \mathbf{def}(w_1^{E_1}) = d_1 \{ \widetilde{m : w} \}_1^{E_1} \quad \mathbf{def}(w_2^{E_2}) = d_2 \{ \widetilde{m : w} \}_2^{E_2} \\ \forall m':w'_2 \in \{ \widetilde{m : w} \}_2 \ . \ \exists w'_1 \ . \ m':w'_1 \in \{ \widetilde{m : w} \}_1 \ \wedge \ \Sigma, (w_1^{E_1}, w_2^{E_2}) \vdash w'_1^{E_1} \leq_{S_{alg}} w'_2^{E_2} \\ d_2 = \mathbf{drop} \implies d_1 = \mathbf{drop} \end{array}}{\Sigma \vdash w_1^{E_1} \leq_{S_{alg}} w_2^{E_2}} \text{INPUT}$$

$$\frac{\begin{array}{c} \mathbf{def}(w_1^{E_1}) = \langle \widetilde{o : u} \rangle_1^{E_1} \quad \mathbf{def}(w_2^{E_2}) = \langle \widetilde{o : u} \rangle_2^{E_2} \\ \forall o':u_1 \in \langle \widetilde{o : u} \rangle_1 \ . \ \exists u_2 \ . \ o':u_2 \in \langle \widetilde{o : u} \rangle_2 \ \wedge \ \Sigma, (w_1^{E_1}, w_2^{E_2}) \vdash u_1^{E_1} \leq_{S_{alg}} u_2^{E_2} \end{array}}{\Sigma \vdash w_1^{E_1} \leq_{S_{alg}} w_2^{E_2}} \text{OUTPUT}$$

► **Theorem 8** (Algorithm completeness). *If $w_1^{E_1} \leq_S w_2^{E_2}$ then $\emptyset \vdash w_1^{E_1} \leq_{S_{alg}} w_2^{E_2}$.*

► **Theorem 9** (Algorithm soundness). *If $\emptyset \vdash w_1^{E_1} \leq_{S_{alg}} w_2^{E_2}$ then $w_1^{E_1} \leq_S w_2^{E_2}$.*

2.2 Types

To statically track the possible typestates an object might be in, we combine them in union types. We also combine them in intersection types to describe combined behaviour from both typestates in the intersection. Their usefulness will be made clearer when we see the result of upcasting a type. Our type hierarchy is a lattice, thus supporting \top and \perp types. Note that types do not include class information. *Typestate Trees* will be used for that (Section 4).

► **Definition 10** (Types grammar). *We call types, ranged over by meta-variable t , the terms generated by the following grammar. Recall that u refers to typestate terms (Definition 1).*

$$t ::= t \cup t \mid t \cap t \mid u^{\widetilde{E}} \mid \top \mid \perp$$

For example, the union type $\text{COMF_ON}^{E_{\text{SUV}}} \cup \text{SPORT_ON}^{E_{\text{SUV}}}$ describes an object that might be in typestate COMF_ON or SPORT_ON .

Let \mathcal{T} be the set of types produced by rule t . Now we need to define a subtyping notion to apply to types. The setting is inspired in work by Muehlboeck and Tate [27], in particular, their definition of reductive subtyping.

► **Definition 11** (Subtyping on types). *Let $\leq \subseteq \mathcal{T} \times \mathcal{T}$ be the relation defined by the following inductive rules.*

$$\frac{}{t \leq \top} \text{TOP} \quad \frac{}{\perp \leq t} \text{BOT} \quad \frac{u_1^{E_1} \leq_S u_2^{E_2}}{u_1^{E_1} \leq u_2^{E_2}} \text{TYPESTATES}$$

$$\frac{t \leq t_i}{t \leq t_1 \cup t_2} \text{UNION_R} \ (i \in \{1, 2\}) \quad \frac{t_i \leq t}{t_1 \cap t_2 \leq t} \text{INTERSECTION_L} \ (i \in \{1, 2\})$$

$$\frac{t_1 \leq t \quad t_2 \leq t}{t_1 \cup t_2 \leq t} \text{UNION_L} \quad \frac{t \leq t_1 \quad t \leq t_2}{t \leq t_1 \cap t_2} \text{INTERSECTION_R}$$

As a sanity check, we show basic subtyping properties on types: reflexivity and transitivity.

► **Lemma 12** (Reflexivity). *For all t , then $t \leq t$.*

► **Lemma 13** (Transitivity). *For all t, t', t'' , if $t \leq t'$ and $t' \leq t''$, then $t \leq t''$.*

An algorithm to check that two types are in a subtyping relationship (i.e., $t \leq t'$) can be implemented by proof search on the inference rules in Def. 11. For these, one can observe that the combined syntactic height of the two types being tested always decreases [27]. Therefore, every recursive search path is guaranteed to always reach a point in which both types being compared are typestates $u \in \mathcal{U}$. Since the algorithm to test $u_1^{E_1} \leq_S u_2^{E_2}$ terminates, the overall algorithm to check subtyping also terminates. For example, it is easy to check that $\text{COMF_ON}^{E_{\text{SUV}}} \leq \text{COMF_ON}^{E_{\text{SUV}}} \cup \text{SPORT_ON}^{E_{\text{SUV}}}$, using the UNION_R rule in Def. 11.

3 Basic operations on types

In this section, we start by describing some preliminary assumptions on the hierarchy of classes, and then proceed to present the three main operations on types performed during type checking: `upcast` (Section 3.1), `downcast` (Section 3.2), and `evolve` (Section 3.3). To showcase these, we use the code in List. 6 that creates an object of type `SUV`, calls the method `turnOn`, switches mode and finally passes the object to method `setSpeed` (lines 3-6).

► **Listing 6** ClientCode class.

```

1  public class ClientCode {
2      public static void example() {
3          SUV suv = new SUV();
4          while (!suv.turnOn()) { System.out.println("turning on..."); }
5          suv.switchMode();
6          setSpeed(suv);
7      }
8      private static void setSpeed(@jatyc.lib.Requires("ON") Car car) {
9          if (car instanceof SUV && ((SUV) car).switchMode() == Mode.SPORT)
10             ((SUV) car).setFourWheels(true);
11             car.setSpeed(50);
12             car.turnOff();
13     }
14 }
```

The method `setSpeed` takes a `Car` in the `ON` state, enforced by the `@Requires` annotation (line 8). The behaviour provided by the `ON` state is also available in `COMF_ON` and `SPORT_ON`, so the method should be prepared to work with a `Car` in the `ON` state or a `SUV` (in `COMF_ON` or `SPORT_ON`). The method tests if the car is a `SUV` and tries to switch to the sport mode (line 9); if it succeeds, it proceeds to set the four wheels drive (line 10). Then, it sets the speed to a given value (line 11) and finishes the protocol by turning off the car (line 12).

Throughout this paper, \mathcal{C} is the set of class names and c is a meta-variable ranging over its elements. Additionally, assume all classes belong to a single-inheritance hierarchy associated.

► **Definition 14** (Super relation on classes). *Super is a partial function such that, given a class c , $\text{Super}(c)$ is the unique direct super class of c , if there is one.*

► **Definition 15** (Subtyping relation on classes). *The relation $\leq_C \subseteq \mathcal{C} \times \mathcal{C}$ is the reflexive and transitive closure of the Super relation.*

With classes and their `Super` relation, we now need to map classes to their corresponding protocols, containing only useful states (i.e., reachable states from the initial one).

► **Definition 16** (Reachable states). *The immediate state reachability relation is a relation over $\mathcal{W} \times \mathcal{W}$, defined as follows: $w'^{\widetilde{E}}$ is immediately reachable from $w^{\widetilde{E}}$, if:*

1. $w = \widehat{d\{m:w\}}$ and $\exists m'. m':w' \text{ in } \widehat{d\{m:w\}}$;
2. $w = \langle o:u \rangle$ and $\exists o'. o':w' \text{ in } \langle o:u \rangle$;
3. $w = s$ and \widetilde{E} includes the equation $s = w'$.

The state reachability relation is the reflexive and transitive closure of the immediate state reachability relation.

Recall that each class c has an associated protocol $s^{\tilde{E}}$, where s is its initial typestate name. We enforce that for any classes c and c' such that $\text{Super}(c') = c$, the protocols of c and c' are subtypes (i.e., the initial typestate of c' is a subtype of the initial typestate of c).

► **Definition 17** (Protocol input states). $\text{ProtInputs}(c)$ is the set of all input states $d\{\widetilde{m : w}\}$ that are reachable from protocol $s^{\tilde{E}}$ of class c .

By only considering reachable input states from the initial typestate name of the protocol, we perform an optimisation that avoids dealing with useless typestates.

To refer to the typestates occurring in a type, we introduce a dedicated auxiliary function.

► **Definition 18** (Typestates in a type). Function $\text{typestates} : \mathcal{T} \rightarrow \mathcal{P}(U)$ is such that, given a type $t \in \mathcal{T}$, $\text{typestates}(t)$ yields the set of typestates occurring in t . Formally,

$$\text{typestates}(t) = \begin{cases} \text{typestates}(t_1) \cup \text{typestates}(t_2) & \text{if } t = t_1 \cup t_2 \text{ or } t = t_1 \cap t_2 \\ \{t\} & \text{if } t \in \mathcal{U} \\ \{\} & \text{if } t = \top \text{ or } t = \perp \end{cases}$$

3.1 Upcast

To upcast a typestate from class c to class c' , we take all typestates in the protocol of c' that are supertypes of the original typestate, and combine them in an intersection type, combining behaviour from different types. If no supertypes are found, the “empty intersection” yields \top , signalling an error.⁴ Since we take supertypes, upcast builds a new type that is a supertype of the original (guaranteed by Theorem 21); and because we intersect the supertypes, we build the most “precise” type possible with typestates in c' (guaranteed by Theorem 22).

► **Definition 19** (Upcast on types). Function $\text{upcast} : \mathcal{T} \times \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{T}$ is such that, given a type t , a class c whose protocol the typestates in t belong to, and a class c' we want to upcast to; $\text{upcast}(t, c, c')$ yields the type obtained by taking the intersection of all supertypes (in the protocol of class c') of typestates included in t . The domain of upcast only includes triples (t, c, c') such that $\text{typestates}(t) \subseteq \text{ProtInputs}(c)$ and $c \leq_C c'$. Formally,

$$\text{upcast}(t, c, c') = \begin{cases} \text{upcast}(t_1, c, c') \cup \text{upcast}(t_2, c, c') & \text{if } t = t_1 \cup t_2 \\ \text{upcast}(t_1, c, c') \cap \text{upcast}(t_2, c, c') & \text{if } t = t_1 \cap t_2 \\ \bigcap \{u' \in \text{ProtInputs}(c') \mid t \leq u'\} & \text{if } t \in \mathcal{U} \\ t & \text{if } t = \top \text{ or } t = \perp \end{cases}$$

To see how upcast works, consider the `setSpeed` call in List. 6. In line 6, after calling `switchMode`, the type of `suv` is `COMF_ON ∪ SPORT_ON` (since we ignore the output of `switchMode`, we do not know the actual typestate). To compute the type of the object passed as parameter, we use the `upcast` function, using as input: (i) `COMF_ON ∪ SPORT_ON` as the type to be upcast; (ii) `SUV` as the starting class; (iii) `Car` as the target class. Since the given type is a union type composed by two elements, the `upcast` function initially unfolds it and creates one intersection for each element (i.e., `COMF_ON` and `SPORT_ON`) containing all their supertypes. In this case, there is just one supertype for each: `ON`. Thus,

$$\text{upcast}(\text{COMF_ON} \cup \text{SPORT_ON}, \text{SUV}, \text{Car}) = \text{ON} \cup \text{ON} = \text{ON} .$$

⁴ In general, upcast operations are always possible, since they produce a supertype of the original type. The issue here is that no operations are safely allowed on \top , so in practise, even if an error is not immediately reported on upcast, there will be an error when trying to use an object with \top type.

As a sanity check, we show that `upcast` builds a type where the typestates composing it belong to the class we upcast to. Recall that Def. 19 has constraints $\text{typestates}(t) \subseteq \text{ProtInputs}(c)$ and $c \leq_C c'$ (the following results assume them). To improve readability we omit stating the constraints explicitly and simply quantify universally types and classes.

► **Lemma 20** (Upcast preserves protocol membership). *For all t , c and c' , then*

$$\text{typestates}(\text{upcast}(t, c, c')) \subseteq \text{ProtInputs}(c').$$

To ensure `upcast` correctness, we show that the result: (i) is a supertype of the given type (Theorem 21); (ii) is the “closest” type to the original with typestates in the protocol of the target class (Theorem 22); and (iii) preserves the subtyping relation (Theorem 23), i.e., `upcast` on types in a subtyping relation produces types that are still in such relation.

► **Theorem 21** (Upcast Consistency). *For all t , c and c' , we have $t \leq \text{upcast}(t, c, c')$.*

► **Theorem 22** (Upcast Least Upper Bound). *For all t , t' , c and c' , such that $\text{typestates}(t') \subseteq \text{ProtInputs}(c')$ and $t \leq t'$, we have $\text{upcast}(t, c, c') \leq t'$.*

► **Theorem 23** (Upcast Preserves Subtyping). *For all t , t' , c and c' , such that $t \leq t'$, we have $\text{upcast}(t, c, c') \leq \text{upcast}(t', c, c')$.*

3.2 Downcast

To downcast a typestate from class c to c' , we take all typestates in the protocol of c' that are subtypes of the original typestate, and combine them in a union type. We use a union type because we need to account for all possible typestates an object might be in. Since we take the subtypes, downcast builds a new type that is a subtype of the original one (guarantee given by Theorem 26); and because we make the union of them, we build the “closest” type possible with typestates in c' (guarantee given by Theorem 27).

► **Definition 24** (Downcast on types). *Function `downcast` : $\mathcal{T} \times \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{T}$ is such that, given a type t , the class c whose protocol the typestates in t belong to, and the class c' we want to downcast to; $\text{downcast}(t, c, c')$ yields the type obtained by taking the union of all subtypes (in the protocol of class c') of typestates included in t . The domain of `downcast` only includes triples (t, c, c') such that $\text{typestates}(t) \subseteq \text{ProtInputs}(c)$ and $c' \leq_C c$. Formally,*

$$\text{downcast}(t, c, c') = \begin{cases} \text{downcast}(t_1, c, c') \cup \text{downcast}(t_2, c, c') & \text{if } t = t_1 \cup t_2 \\ \text{downcast}(t_1, c, c') \cap \text{downcast}(t_2, c, c') & \text{if } t = t_1 \cap t_2 \\ \bigcup \{u' \in \text{ProtInputs}(c') \mid u' \leq t\} & \text{if } t \in \mathcal{U} \\ t & \text{if } t = \top \text{ or } t = \perp \end{cases}$$

Note that `downcast` only yields \perp if given \perp . Consider the third case of Def. 24. The union only yields \perp if no sub-typestates in the protocol of c' are found. But that is impossible. If we downcast from a typestate t (in c) to a subclass c' , and since the protocol of c' is a subtype of the one of c , there will necessarily be at least one typestate in c' subtype of t . Moreover, Theorem 54 will show that our overall approach is sound.

To see how `downcast` works, consider the downcast performed in line 9 of List. 6. To compute the type of (SUV) `car`, we use `downcast`, defined in Def. 24, passing as parameter: (i) `ON` as the type to be downcast (given the `Requires` annotation); (ii) `Car` as the starting class; (iii) `SUV` as the target class. Since the type passed as parameter is a simple typestate, the `downcast` function just creates a union containing all the subtypes of `ON`. Concretely,

$$\text{downcast}(\text{ON}, \text{Car}, \text{SUV}) = \text{COMF_ON} \cup \text{SPORT_ON} .$$

As a sanity check, we show that `downcast` builds a type where the typestates composing it belong to the class we downcast to. Recall that Def. 24 has constraints $\text{typestates}(t) \subseteq \text{ProtInputs}(c)$ and $c' \leq_C c$. (the following results assume them). To improve readability, the constraints are implicit and we simply quantify universally types and classes.

► **Lemma 25** (Downcast preserves protocol membership). *For all t, c and c' , we have*

$$\text{typestates}(\text{downcast}(t, c, c')) \subseteq \text{ProtInputs}(c').$$

To ensure `downcast` correctness, we show that the result: (i) is a subtype of the given type (Theorem 26); (ii) is the “closest” type to the original with typestates in the protocol of the target class (Theorem 27); and (iii) preserves the subtyping relation i.e., `downcast` on types in a subtyping relation produces types that are still in such relation (Theorem 28).

► **Theorem 26** (Downcast Consistency). *For all t, c and c' , we have $\text{downcast}(t, c, c') \leq t$.*

► **Theorem 27** (Downcast Greatest Lower Bound). *For all t, t', c and c' , such that $\text{typestates}(t') \subseteq \text{ProtInputs}(c')$ and $t' \leq t$, we have $t' \leq \text{downcast}(t, c, c')$.*

► **Theorem 28** (Downcast Preserves Subtyping). *For all t, t', c and c' , such that $t \leq t'$, we have $\text{downcast}(t, c, c') \leq \text{downcast}(t', c, c')$.*

Additionally, we relate the result of upcasting and then downcasting with the original type, as well as, the result of downcasting and then upcasting. The first follows from Theorems 21 and 27, the second from Theorems 22 and 26. These corollaries are also important to ensure the soundness of the approach (Theorem 54).

► **Corollary 29** (Downcast reverses upcast). *For all t, c and c' , we have*

$$t \leq \text{downcast}(\text{upcast}(t, c, c'), c', c).$$

► **Corollary 30** (Upcast reverses downcast). *For all t, c and c' , we have*

$$\text{upcast}(\text{downcast}(t, c, c'), c', c) \leq t.$$

3.3 Evolve

Whenever we perform a method call on an object with a given type, we need to compute the new type representing the typestates the object might be in after the call. To compute such type and rule out misconduct, we define the `evolve` function, which yields \top when a method is not callable in the given type. Ret_m is the set of outputs returnable by method m .

► **Definition 31** (Evolve). *Function $\text{evolve} : \mathcal{T} \times \mathcal{M} \times \mathcal{O} \rightarrow \mathcal{T}$ is such that, given a type t , a method m , and an object $o \in \text{Ret}_m$; $\text{evolve}(t, m, o)$ yields the new type obtained by executing m on any object currently with type t , where o is the value returned by m . Its definition relies on the auxiliary functions $\text{evolveU} : \mathcal{U} \times \mathcal{M} \times \mathcal{O} \rightarrow \mathcal{U}$ and $\text{evolveY} : \mathcal{Y} \times \mathcal{O} \rightarrow \mathcal{U}$. Formally,*

$$\text{evolve}(t, m, o) = \begin{cases} \text{evolve}(t_1, m, o) \cup \text{evolve}(t_2, m, o) & \text{if } t = t_1 \cup t_2 \\ \text{evolve}(t_1, m, o) \cap \text{evolve}(t_2, m, o) & \text{if } t = t_1 \cap t_2 \\ \text{evolveU}(t, m, o) & \text{if } t \in \mathcal{U} \\ t & \text{if } t = \top \text{ or } t = \perp \end{cases}$$

$$\text{evolveU}(u, m, o) = \begin{cases} w & \text{if } \text{def}(u) = {}_d\{m : w, \widetilde{m : w}\} \wedge w \in \mathcal{U} \\ \text{evolveY}(w, o) & \text{if } \text{def}(u) = {}_d\{m : w, \widetilde{m : w}\} \wedge w \in \mathcal{Y} \\ \top & \text{otherwise} \end{cases}$$

$$\text{evolveY}(y, o) = \begin{cases} u & y = \langle o : u, \widetilde{o : u} \rangle \\ \perp & \text{otherwise} \end{cases}$$

Since `evolve` is deterministic, it is defined as a function, not as a labelled transition system.

To see how `evolve` works, consider the `switchMode` call in line 9 of List. 6. To compute the type of `car`, we use `evolve`, defined in Def. 31, passing as parameter: (i) the type `COMF_ON ∪ SPORT_ON` be evolved (given the result of the `downcast` function); (ii) the method `switchMode` to make the type evolve; (iii) the expected output `Mode.SPORT` to enter the `if` branch. Since the type, passed as parameter, is a union type composed by two elements, the `evolve` function is called recursively, and then the auxiliary function `evolveU` is called for `COMF_ON` and `SPORT_ON`. Since the `switchMode` action leads to an output state, for both `COMF_ON` and `SPORT_ON` (see List. 2), the auxiliary function `evolveY` is invoked. Concretely,

$$\text{evolve}(\text{COMF_ON} \cup \text{SPORT_ON}, \text{switchMode}, \text{Mode.SPORT}) = \text{SPORT_ON} \cup \text{SPORT_ON}$$

that can be simplified to `SPORT_ON`. Notice that the evolved type has the same structure of the one before calling the `evolve` function i.e., a union type.

As a sanity check, we show that `evolve` produces a type containing only typestates belonging to the initial class.

► **Lemma 32** (`Evolve preserves protocol membership`). *For all t, m, o, c ,*

$$\text{typestates}(t) \subseteq \text{ProtInputs}(c) \text{ implies } \text{typestates}(\text{evolve}(t, m, o)) \subseteq \text{ProtInputs}(c)$$

To ensure `evolve` correctness, we show that the result preserves the subtyping relation: `evolve` on types in a subtyping relation produces types that still are in such relation.

► **Theorem 33** (`Evolve preserves subtyping`). *For all t and t' such that $t \leq t'$, we have that*

$$\text{evolve}(t, m, o) \leq \text{evolve}(t', m, o).$$

We also relate `evolve` with `upcast` and `downcast` showing that: (i) `upcast` after `evolve` produces a subtype of the inverse sequence of operations (Theorem 34); and (ii) `downcast` after `evolve` produces a supertype of the inverse sequence of operations (Theorem 35). These theorems are key to ensure soundness (Theorem 54). For readability, we omit the constraints on the universally quantified variables needed to use `upcast` and `downcast`.

► **Theorem 34** (`Evolve and upcast`). *For all t, m, o, c and c' , we have that*

$$\text{upcast}(\text{evolve}(t, m, o), c, c') \leq \text{evolve}(\text{upcast}(t, c, c'), m, o).$$

► **Theorem 35** (`Evolve and downcast`). *For all t, m, o, c and c' , we have that*

$$\text{evolve}(\text{downcast}(t, c, c'), m, o) \leq \text{downcast}(\text{evolve}(t, m, o), c, c')$$

4 Typestate Trees

In this section, we describe *Typestate Trees*, the crucial data structure we use to solve the problem of casting in the middle of a protocol. These trees associate classes with types containing only states in the protocol of those classes (i.e., $\text{typestates}(t) \subseteq \text{ProtInputs}(c)$). The tree root indicates the static type of a variable and the corresponding type (in \mathcal{T}) at a given program point. All other nodes describe what should be the type if we downcast to the corresponding class. The type in the root is always a sound approximation of the runtime execution. The types in other nodes are also sound only if the object is an instance of the corresponding class. This will imply that type safety is guaranteed up-to class downcasts being performed to a class of which an object is a subtype of. Hereafter we define well-formed typestate trees and auxiliary functions. Sections 4.1, 4.2, 4.3, and 4.4, describe the main operations on typestate trees: `upcastTT`, `downcastTT`, `evolveTT`, and `mrgTT`, respectively.

► **Definition 36** (Typestate Trees). *Recall that c ranges over classes (\mathcal{C}) and t ranges over types (\mathcal{T}). Let \mathcal{TT} be the smallest set of triples satisfying the following rules:*

$$\frac{n \geq 1 \quad \forall i, 1 \leq i \leq n . \ tt_i \in \mathcal{TT}}{(c, t, \{ \}) \in \mathcal{TT}} \quad \frac{}{(c, t, \{ tt_i \mid 1 \leq i \leq n \}) \in \mathcal{TT}}$$

Notice that triples in \mathcal{TT} represent trees and are composed of: the class c and the type t of the root, and a set of subtrees (again triples in \mathcal{TT}), one for each root child. Such a set is empty if the tree root has no children (i.e., the tree simply represents a leaf). Throughout this paper, tt ranges over elements of \mathcal{TT} and tts ranges over sets of elements of \mathcal{TT} . We need functions to destroy an element of \mathcal{TT} (which is a triple). Let $\text{cl}((c, t, tts)) = c$, $\text{ty}((c, t, tts)) = t$, and $\text{children}((c, t, tts)) = tts$.

► **Definition 37** (No duplicate classes). *The predicate `nodup` over $\mathcal{P}(\mathcal{TT})$ asserts that, given a set $tts \in \mathcal{P}(\mathcal{TT})$, no two typestates trees in tts have the same associated class. Formally, `nodup(tts)` holds if: $\forall tt, tt' \in tts . \ \text{cl}(tt) = \text{cl}(tt') \Rightarrow tt = tt'$.*

► **Definition 38** (Well-formedness Of Typestate Trees). *The predicate `⊤` over \mathcal{TT} asserts that, given a typestate tree (c, t, tts) , it is correctly constructed. Formally,*

$$\frac{\text{typestates}(t) \subseteq \text{ProtInputs}(c) \quad \text{nodup}(tts) \quad \forall tt \in tts . \ \text{Super}(\text{cl}(tt)) = c \wedge \text{upcast}(\text{ty}(tt), \text{cl}(tt), c) \leq t \wedge \vdash tt}{\vdash (c, t, tts)}$$

So, a typestate tree (c, t, tts) is well-formed under the following conditions: (i) all the typestates of type t belong to the protocol of class c ; (ii) there are no two children with the same class; (iii) the classes associated with each child tree are direct subclasses of c ; (iv) if we upcast a type of a child tree, we get a subtype of t ; and (v) each child is also well-formed. Condition (iv) ensures that the type of a child tree is never less “precise” than the type of the parent. From now on, we only consider well-formed typestate trees.

To illustrate the concept, suppose that in line 3 of List. 6, instead of assigning the newly created object to a `SUV` variable, we assign it to a `Car` one, performing an upcast. Since the static and actual type are different, we need a typestate tree to handle future casts. Given Def. 36 and Def. 38, the resulting typestate tree is $(\text{Car}, \text{OFF}, \{(\text{SUV}, \text{OFF}, \{ \})\})$.

4.1 Upcast

Upcasting a typestate tree to class c' ensures that the resulting root class is c' , by recursively following the `Super` relation and building up new tree roots until the root class is c' .

► **Definition 39** (Upcast on typestate trees). Function $\text{upcastTT} : \mathcal{T}\mathcal{T} \times \mathcal{C} \rightarrow \mathcal{T}\mathcal{T}$ is such that $\text{upcastTT}((c, t, tts), c')$ performs an upcast on typestate tree (c, t, tts) to class c' . The domain of upcastTT only includes pairs $((c, t, tts), c')$ such that $c \leq_C c'$. Formally,

$$\text{upcastTT}((c, t, tts), c') = \begin{cases} (c, t, tts) & \text{if } c = c' \\ \text{upcastTT}((\text{Super}(c), \text{upcast}(t, c, \text{Super}(c)), \{(c, t, tts)\}), c') & \text{otherwise} \end{cases}$$

Notice that, under the assumption on the domain of the upcastTT , the function terminates since the distance between c and c' decreases with each recursive step.

► **Theorem 40** (Typestate Trees Well-formedness Preserved By Upcast). For all c'', tt , such that $\vdash tt$ and $\text{cl}(tt) \leq_C c''$, it holds that $\vdash \text{upcastTT}(tt, c'')$.

To see how upcastTT works, consider the `setSpeed` call in List. 6. In line 8, after calling `switchMode`, the object `suv` has the following typestate tree $(\text{SUV}, \text{COMF_ON} \cup \text{SPORT_ON}, \{\})$. When passing `suv` to `setSpeed`, we need to upcast from `SUV` to `Car`. To do that, we use the upcastTT function, defined in Def. 39, passing as parameter: (i) $(\text{SUV}, \text{COMF_ON} \cup \text{SPORT_ON}, \{\})$ as the typestate tree to be upcast; and (ii) `Car` as the target class. Thus,

$$\text{upcastTT}((\text{SUV}, \text{COMF_ON} \cup \text{SPORT_ON}, \{\}), \text{Car}) = (\text{Car}, \text{ON}, \{(\text{SUV}, \text{COMF_ON} \cup \text{SPORT_ON}, \{\})\}).$$

It is crucial to notice that to upcast a typestate tree, we must perform multiple upcasts, incrementally building up new tree roots, not only to preserve the well-formedness property, but also to ensure soundness. For readability sake, we show the problem with an abstract, but simple example. Take classes `A`, `B` and `C`, where $\text{Super}(C) = B$, $\text{Super}(B) = A$, and the protocol equations associated with each class listed below. Recall that `end = drop{}`.

$$\begin{array}{ll} A1 = \{ m1 : end \} & C1 = \{ m1 : end, m2 : end, m3 : C2 \} \\ B1 = \{ m1 : end, m2 : end \} & C2 = \{ m1 : end, m4 : end \} \end{array}$$

Given the protocols above and according to Def. 4 we have:

$$C1 \leq_S B1 \leq_S A1 \text{ and } C2 \leq_S A1, \text{ but } C2 \not\leq_S B1.$$

`C2` not being a subtype of `B1` is not a problem *per se*, but it may be when upcasting, if we define it to go directly to the root instead of going level-by-level, as downcasting after upcasting should lead to the original state.⁵ To see that, consider the code in List. 7, which contains an unsafe method call, but would be accepted. At first, we create an object `c` of class `C` and we call its method `m3`, producing a new typestate, i.e., `C2`. Then, we assign `c` to variable `a` performing an upcast from class `C` to `A` (and from typestate `C2` to `A1`). We finally perform a sequence of downcasts on `a` leading the object to class `C` (and to typestate `C1`).

Listing 7 Direct upcast example.

```

1 C c = new C(); // C1
2 c.m3(); // C2
3 A a = c; // A1: unsound upcast!
4 B b = (B) a; // B1: downcast level-by-level
5 C c = (C) b; // C1: incorrect! the state should be C2 (that of line 2)
6 c.m2(); // unsafe!
```

⁵ Technically, downcasting after upcasting returns an over-approximation of the original state.

In detail (for those interested), notice that the result of upcasting $C2$ directly to class A (line 3, List. 7) is $A1$, since it is the only supertype of $C2$, i.e., $C2 \leq_S A1$. To downcast $A1$ to class B , we check all the typestates in the protocol of B subtypes of $A1$. Since only $B1$ is subtype of $A1$, that is the downcast result (line 4). Similarly, since only $C1$ is a subtype of $B1$, it is the result of downcasting from $B1$ (line 5). Notice how a direct upcast to A , followed by a downcast to B , and then to C , results in a different typestate with respect to the initial one. This is unsound: $C1$ and $C2$ are unrelated. The issue is that a direct upcast to A makes us lose the information about $C1$ not having supertypes among typestates in B . Since we first upcast $C2$ to B , getting \top as result, we find out that $C2$ has no supertypes among typestates in B . Additionally, since we use typestate trees, downcasting to C leads back to $C2$.

4.2 Downcast

When downcasting a given typestate tree tt to class c , we ensure that the root class of the resulting tree is c . If we find a subtree in tt whose class is c , we pick it as the result (by well-formedness, it is unique). Otherwise, we build a new tree downcasting from the most “precise” type information in tt . For this, we use the `closestSubT` function to look for the subtree whose class is hierarchically the “closest” to c .

► **Definition 41** (Closest subtree). *The function `closestSubT` : $\mathcal{TT} \times \mathcal{C} \rightarrow \mathcal{TT}$ is such that $\text{closestSubT}(tt, c)$ yields the subtree associated with the closest superclass of c occurring in tt . The domain of `closestSubT` only includes pairs (tt, c) such that $c \leq_C \text{cl}(tt)$. Formally,*

$$\text{closestSubT}(tt, c) = \begin{cases} \text{closestSubT}(tt', c) & \text{if } c \leq_C \text{cl}(tt') \wedge tt' \in \text{children}(tt) \\ tt & \text{otherwise} \end{cases}$$

To illustrate the use of `closestSubT`, consider classes A , B , and C , where $\text{Super}(B) = A$ and $\text{Super}(C) = B$. Let tt be $(A, t, \{(B, t', \{\})\})$. Then the following equalities hold: $\text{closestSubT}(tt, A) = tt$; $\text{closestSubT}(tt, B) = (B, t', \{\})$; and $\text{closestSubT}(tt, C) = (B, t', \{\})$. The first two cases are easy to understand: the function yields the subtree whose class is precisely the one we are looking for. In the third case, since there is no subtree in tt whose class is C , $\text{closestSubT}(tt, C)$ yields the subtree corresponding to B , which is the “closest” superclass of C present in tt , i.e., $(B, t', \{\})$. Now, suppose instead that $\text{Super}(B) = A$ and $\text{Super}(C) = A$ (i.e., B and C are “siblings”). Then $\text{closestSubT}(tt, C)$ would yield the entire tree tt whose class is A , which is the “closest” superclass of C present in tt . Lemma 42 ensures the correctness of `closestSubT` and is useful for the soundness proof (Theorem 54).

► **Lemma 42** (Closest correctness). *For all tt and c , if $c \leq_C \text{cl}(tt)$ then*

$$c \leq_C \text{cl}(\text{closestSubT}(tt, c)).$$

► **Definition 43** (Downcast on typestate trees). *Function `downcastTT` : $\mathcal{TT} \times \mathcal{C} \rightarrow \mathcal{TT}$ is such that $\text{downcastTT}(tt, c)$ performs a downcast on typestate tree tt to class c . The domain of `downcastTT` only includes pairs (tt, c) such that $c \leq_C \text{cl}(tt)$. Formally,*

$$\text{downcastTT}(tt, c) =$$

$$\begin{cases} tt' & \text{if } tt' = \text{closestSubT}(c, tt) \wedge c = \text{cl}(tt') \\ (c, \text{downcast}(\text{ty}(tt'), \text{cl}(tt'), c), \{\}) & \text{otherwise } tt' = \text{closestSubT}(c, tt) \end{cases}$$

► **Theorem 44** (Typestate Trees Well-formedness Preserved By Downcast). *For all c, tt , such that $\vdash tt$ and $c \leq_C \text{cl}(tt)$, it holds that $\vdash \text{downcastTT}(tt, c)$.*

To see how `downcastTT` works, observe how `(SUV) car` would be checked (in List. 6). To compute its typestate tree, we use `downcastTT`, defined in Def. 43, passing as parameter: (i) `(Car, ON, {})` as the typestate tree to downcast; (ii) `SUV` as the target class. Notice that, in the case the root is also a leaf, we need to replace it with the result of `downcastTT`. Concretely, $\text{downcastTT}((\text{Car}, \text{ON}, \{}), \text{SUV}) = (\text{SUV}, \text{COMF_ON} \cup \text{SPORT_ON}, \{})$.

4.3 Evolve

To compute the typestate tree of an object after a call, we define the `evolveTT` function.

► **Definition 45** (Evolve on typestate trees). *Function `evolveTT : TT × M × O → TT` is such that `evolveTT(tt, m, o)` yields a new typestate tree resulting from applying `evolve(t, m, o)` (Def. 31) to all the nodes of tt . The domain of `evolveTT` only includes triples (tt, m, o) such that $o \in \text{Ret}_m$ (i.e. the set of outputs returnable by method m). Formally,*

$$\text{evolveTT}((c, t, tts), m, o) = (c, \text{evolve}(t, m, o), \bigcup_{tt_i \in tts} \text{evolveTT}(tt_i, m, o)).$$

Notice that, when the set tts is empty, $\text{evolveTT}((c, t, tts), m, o) = (c, \text{evolve}(t, m, o), \{\})$

► **Theorem 46** (Typestate Trees Well-formedness Preserved By Evolve). *For all tt, m, o , such that $\vdash tt$, it holds that $\vdash \text{evolveTT}(tt, m, o)$.*

■ **Listing 8** `EvolveTT` example.

```
1 Car c = new SUV();
2 if (c.turnOn()) c.turnOff();
```

To see how `evolveTT` works, consider the code presented in List. 8 (where the protocols of `Car` and `SUV` are presented in List. 1 and List. 2). The typestate tree of `c` is `(Car, OFF, {(SUV, OFF, {})})`. When the `turnOn` call occurs, we need to “evolve” each node of the typestate tree. To compute the resulting tree, we use `evolveTT`, defined in Def. 45, passing as parameter: (i) the typestate tree of `c`; (ii) `turnOn` as the method called; (iii) `true` as the expected output to enter the `if` branch. Concretely,

$$\text{evolveTT}((\text{Car}, \text{OFF}, \{(SUV, \text{OFF}, \{\})\}), \text{turnOn}, \text{true}) = (\text{Car}, \text{ON}, \{(SUV, \text{ON}, \{\})\}).$$

Notice that every node of the typestate tree is “evolved” using the `evolve` function.

4.4 Merge

In the case of branching code, one has to merge type information coming from all different branches, so that subsequent code can be properly analysed by considering all possibilities (e.g., merging type information coming from both branches of an `if` statement). To this end, we define the `mrgTT` function, which merges two typestate trees. Before presenting `mrgTT`, we define some auxiliary functions, crucial for the formalisation.

► **Definition 47.** *Function `height : P(TT) → N` is such that `height(tt)` yields the greatest number of nodes traversed, in tt , from the root to one of the leaves (both included).*

► **Definition 48.** *Function `clss : P(TT) → P(C)` is such that `clss(tts)` yields the set of classes associated with the typestate trees in tts . Formally, $\text{clss}(tts) = \{\text{cl}(tt) \mid tt \in tts\}$.*

► **Definition 49.** Function $\text{find} : \mathcal{C} \times \mathcal{P}(\mathcal{T}\mathcal{T}) \rightarrow \mathcal{T}\mathcal{T}$ is such that, given a class c and set of typestate trees tts with $c \in \text{clss}(tts)$ and $\text{nodup}(tts)$, $\text{find}(c, tts)$ yields the unique typestate tree in set tts whose class is c .

► **Definition 50 (Merge).** Function $\text{mrgTT} : \mathcal{T}\mathcal{T} \times \mathcal{T}\mathcal{T} \rightarrow \mathcal{T}\mathcal{T}$ is such that, given typestate trees tt and tt' , $\text{mrgTT}(tt, tt')$ yields the typestate tree obtained by merging tt and tt' . The domain of mrgTT only includes pairs (tt, tt') such that $\text{cl}(tt) = \text{cl}(tt')$. Formally,

$$\text{mrgTT}((c, t, tts), (c, t', tts')) = (c, t \cup t', tts_1 \cup tts_2 \cup tts_3)$$

$$\text{where } tts_1 = \bigcup_{c_i \in \text{clss}(tts) \cap \text{clss}(tts')} \text{mrgTT}(\text{find}(c_i, tts), \text{find}(c_i, tts'))$$

$$tts_2 = \bigcup_{c_i \in \text{clss}(tts) \setminus \text{clss}(tts')} \text{mrgTT}(\text{find}(c_i, tts), (c_i, \text{downcast}(t', c, c_i), \{\}))$$

$$tts_3 = \bigcup_{c_i \in \text{clss}(tts') \setminus \text{clss}(tts)} \text{mrgTT}((c_i, \text{downcast}(t, c, c_i), \{\}), \text{find}(c, tts'))$$

Note that mrgTT terminates since $\text{height}(tt) + \text{height}(tt')$ decreases with each recursive step. Moreover, mrgTT is symmetric, i.e., $\text{mrgTT}(tt, tt')$ gives the same result as $\text{mrgTT}(tt', tt)$.

► **Theorem 51 (Typestate Trees Well-formedness Preserved By Merge).** For all tt, tt' , such that $\text{cl}(tt) = \text{cl}(tt')$, $\vdash tt$, and $\vdash tt'$, it holds that $\vdash \text{mrgTT}(tt, tt')$.

To see how mrgTT works, consider the `if` statement in List. 6 (lines 12-14). Notice that, although the `else`-branch is missing, in the process of computing the typestate tree of `car`, we need to consider it to be there (to account for all possible outputs returned by `switchMode`). To compute such typestate tree, we use mrgTT , defined in Def. 50, passing as parameters: (i) `(SUV, SPORT_ON, {})` and (ii) `(SUV, COMF_ON, {})`. Since neither of the parameters have children nodes, it is enough to make the union of the root types. Concretely,

$$\text{mrgTT}((\text{SUV}, \text{SPORT_ON}, \{}), (\text{SUV}, \text{COMF_ON}, \{})) = (\text{SUV}, \text{SPORT_ON} \cup \text{COMF_ON}, \{}).$$

5 Typestate Trees Soundness

In this section, we discuss why we consider type-safe a programming language equipped with our subtypestate mechanism. Such result relies on the key property that given a typestate tree that soundly approximates the current runtime typestate of an object, operations on it result in new typestate trees that still soundly approximate the runtime typestate. This assumes that class downcasts are performed to a class of which the object is a subtype of. So, we do not provide static guarantees that class downcasts will not throw at runtime.

► **Definition 52 (Sequence of upcasts on types).** Function $\text{upcast}^* : \mathcal{T} \times \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{T}$ is such that $\text{upcast}^*(t, c, c')$ performs zero or more upcasts from c to c' step-by-step, following the class hierarchy. The domain of upcast^* only includes triples (t, c, c') such that $\text{typestates}(t) \subseteq \text{ProtInputs}(c)$ and $c \leq_C c'$. Formally,

$$\text{upcast}^*(t, c, c') = \begin{cases} t & \text{if } c = c' \\ \text{upcast}^*(\text{upcast}(t, c, \text{Super}(c)), \text{Super}(c), c') & \text{otherwise} \end{cases}$$

Since the distance between c and c' decreases with each recursive step, upcast^* terminates.

The next relation describes a typestate tree where type information is sound with respect to class c' and type t' . That is, assuming c' and t' represent the exact runtime type of a given object, a sound typestate tree correctly approximates such type. Note that the root has to be necessarily sound with respect to runtime, while the other nodes only need to be sound if the initialising class of the object is a subclass of the class associated with that node. Thus, all non-root nodes describe the type of the object if indeed the object is an instance of the corresponding class. This implies that if we downcast, possibly turning a non-root node into the new root, we preserve soundness only if the runtime downcast succeeds.

► **Definition 53** (Soundness Of Typestate Trees). *The predicate $\vdash_{c',t'}$ over \mathcal{TT} , with $\text{typestates}(t') \subseteq \text{ProtInputs}(c')$, asserts that, given a (well-formed) typestate tree (c, t, tts) , it is sound with respect to class c' and type t' . Formally,*

$$\frac{c' \leq_C c \quad \text{upcast}^*(t', c', c) \leq t \quad \forall tt \in tts . \ c' \leq_C \text{cl}(tt) \Rightarrow \vdash_{c',t'} tt}{\vdash_{c',t'} (c, t, tts)}$$

The next theorem shows that soundness is preserved by typestate tree operations. Note that soundness after downcast is only preserved if at runtime the downcast does not throw an exception (thus the assumption $c \leq_C c' \leq_C \text{cl}(tt)$ on the second item of Theorem 54).

► **Theorem 54** (Typestate Trees Soundness Preservation). *Soundness is preserved by:*

upcast – for all c, t, c', tt , such that $\vdash_{c,t} tt$ and $\text{cl}(tt) \leq_C c'$, it holds that

$$\vdash_{c,t} \text{upcastTT}(tt, c')$$

downcast – for all c, t, c', tt , such that $\vdash_{c,t} tt$ and $c \leq_C c' \leq_C \text{cl}(tt)$, it holds that

$$\vdash_{c,t} \text{downcastTT}(tt, c')$$

evolve – for all c, t, tt, m, o , such that $\vdash_{c,t} tt$, it holds that

$$\vdash_{c,\text{evolve}(t,m,o)} \text{evolveTT}(tt, m, o)$$

merge – for all c, t, tt_1, tt_2 , such that $\vdash_{c,t} tt_1$ or $\vdash_{c,t} tt_2$, and $\text{cl}(tt_1) = \text{cl}(tt_2)$, it holds that $\vdash_{c,t} \text{mrgTT}(tt_1, tt_2)$.

Having shown our approach sound, the following section explains how the functions defined in Section 4 are used during type checking.

6 Application to type checking

We believe the setting presented is quite general and applicable to many object-oriented languages. In this section, we explain, how in detail, assuming a common syntax.⁶ We start by describing how declarations are analysed in JaTyC, followed by expressions, and then statements. We use the Kleene star to denote (possibly empty) sequences.

Class declarations and overriding. First, it is crucial to guarantee that protocols are well-formed and the relation between classes and their protocols makes sense. To this, we ensure that all methods mentioned in the protocol are declared in the class. Similarly, we check that all mentioned outputs are return values of the corresponding methods. Additionally, we check typestate input contravariance and output covariance in overridden methods, since these may include `@Requires` and `@Ensures` annotations in parameters and return types, respectively, which limit the typestates received/returned. Finally, we ensure that the subclass protocol is

⁶ Formalising a type checking system for one particular language, mechanising, and proving it sound is a matter for another paper. Doing that for a core Java-like language is work-in-progress.

a subtype of the superclass one (i.e., the initial state of the former is a subtype of the initial state of the latter according to Def. 4). Thanks to these checks, dynamic dispatch works transparently: if a method is callable on a supertype, it is also callable on its subtypes.

To type check a class, we analyse each method following the sequences of calls allowed by the protocol, similarly to the approach by Bravetti et al. [11], so that method analysis benefits from type information coming from the analyses of methods called before. Type information is stored in a map from locations (local variables, fields of the `this` object, and code expressions) to typestate trees (Def. 36). We also store the typestate trees of expressions since these may evaluate to typestate objects, which must be tracked. Moreover, type information of final states is checked to ensure all fields either correspond to a terminated protocol or are aliased (explained later), to ensure *protocol completion* of references in fields.

Method declarations: `@Ensures(s) type m((@Requires(s) type x)*) {st}`. To check a method, we build a control flow graph [1] with the Checker Framework [30]. Then, we traverse it, visiting each expression or statement, and propagate type information. For each expression, we take the type information obtained from analysing the previous one, and produce new information. Expression or statement analysis is described later.

The initial type information (i.e., the initial input of the graph traversal) is composed by the types of the parameters, expressed via the `@Requires` annotation, combined with information about fields (coming from previous method analysis, as explained before). If a `@Requires` annotation is omitted, it means we expect an aliased reference. Return statements are analysed like assignments, while making sure the returned expression type is a subtype of the one declared via the `@Ensures` annotation. If no annotation is provided, we return an aliased reference. At the end of a method body, we ensure variables and code expressions are either aliased or in a final state, guaranteeing *protocol completion*.

Variable declarations or assignments: `[type] x = exp`. To check a variable declaration or assignment, we call `upcastTT` (Def. 39) on the typestate tree of the right-hand-side, and associate the result with the variable (or field) in the left-hand-side. If `upcastTT` yields a typestate tree with \top as root type, the assignment is not allowed and we report an error. Given how the control flow graph is built, the expression on the right-side was already checked when we reach this point. If we override a typestate tree corresponding to a non-terminated protocol, we also report an error, since the assignment may compromise protocol completion of the overridden reference. Assignments may produce aliasing among variables. Since an object’s state could be modified via multiple aliases, we restrict aliasing to allow us to statically track object states. We enforce a linear discipline: only one variable is “active”, while the others are marked as aliased (and cannot call protocol methods). We also mark the right-hand-side expression as aliased when checking a variable declaration or assignment.

Method call expressions: `exp.m(exp*)`. To check a call, we first ensure the receiver expression is not `null`. We can do this because we distinguish between nullable and non-null types. Then, we analyse each *parameter assignment* applying the same rules explained before. This ensures that calls like `obj.m(x,x)` do not create unintended aliases. We also ensure that the root types of the typestate trees associated with the parameter expressions are subtypes of the expected types in the method signature. Following this, we proceed to check the call itself. We ensure the receiver expression is a non-aliased reference and use `evolveTT` (Def. 45) to compute the typestate tree associated with the receiver after the call, passing the current typestate tree, the method name, and a possibly returned output (if the method call appears in an `if` or `switch` statement). Note that `evolveTT` might be called several times to consider all possible outputs. If `evolveTT` yields a typestate tree where the root type is \top , then the method is not available to be called in that state, so we report an error.

Cast expressions: `(C) exp`. When checking a cast, we know that the inner expression was already checked, similarly to what happens to other expressions. To check it, we must use either `upcastTT` (Def. 39) or `downcastTT` (Def. 43), passing the inner expression typestate tree and the target class. We test if we are upcasting or downcasting by comparing the inner expression static type with the target class. The result is associated with the cast expression and the inner one is marked as aliased. As for assignments, if `upcastTT` yields a typestate tree with \top in the root, we report an error. However, `downcastTT` does not produce errors because we provide type safety up-to downcasts not throwing runtime exceptions.

One key detail about cast expressions is that if a cast expression is the receiver object of a method call, after checking the call, the new type of the receiver object is associated with the most inner expression which is not a cast, not with the cast expression itself. For example, if the receiver is `(A) ((B) x)`, the new type information is associated with `x` directly, not with `(A) ((B) x)`, so that `x` can be used again later (instead of being aliased). This will require an upcast to the class of `x`, but no information is lost, thanks to typestate trees.

New expression: `new C(exp*)`. The initialisation of a new object is analysed similarly to a method call (since we are calling the constructor), except that it returns a new object. So, we associate the expression with a typestate tree with only a root: the class is the object type we are constructing, and the type (from Def. 10) is the initial typestate of the protocol.

If statements: `if (exp) { st } else { st' }`. For simplicity, up until now we omitted an implementation detail crucial to type check `if` statements (and `switch` statements): during the control flow graph traversal, we do not simply propagate a map from locations to typestate trees; we keep track of type information depending on the values of other expressions. For instance, to analyse a method call in a condition of an `if`, we track the type information for when the condition is `true` separately from the one when it is `false`. So, when checking an `if`, we just propagate the former to the first branch, and the latter to the second branch. We also make sure to invalidate such “conditional” type information once it is no longer relevant. Finally, the typestate trees associated with each location after the `if` are the result of merging type information from both branches, using `mrgTT` (Def. 50).

Switch statements: `switch (exp) { (case val : st)* }`. We analyse a `switch` statement similarly to an `if` one. A method call in the expression of a `switch` statement produces type information different for each `case`, but we consider enumeration values that may be returned instead of boolean values. So, to check a `switch` statement, we just need to propagate the information that holds when a given `case` is matched to the related branch. Again, we invalidate this “conditional” type information once it is no longer relevant.

While statements: `while (exp) { st }`. While statements are analysed like `if` ones, except the flow graph is different: after the body is executed, execution returns to the condition. Because of this, we might traverse the same expression or statement in the graph more than once. If that occurs, we merge the new gathered information with the previous one. To ensure that the static analysis terminates, we avoid analysing an expression again if no new type information was gathered. This is guaranteed to occur because the number of all possible typestates is finite. In the worst case, when merging, we might produce a union of all typestates. Typestate trees are also finite because the number of classes is finite.

7 Use Cases

To showcase the applicability and expressiveness of our approach, we start by explaining how the code in List. 6 is type checked in detail. Then, we present a suite of examples with polymorphic code⁷, inspired from **cyber-physical systems**, showing that: (i) JaTyC detects errors the standard Java type checker does not detect; (ii) our setting is flexible and expressive enough to model interesting and intricate scenarios.

Type checking List. 6. To type check the `ClientCode` class (which has no protocol), we analyse the static methods `example` and `setSpeed`, independently (since static methods are not part of a class protocol). The list of steps to type check the `example` method follows:

- Check the expression `new SUV()`, associating it with a leaf typestate tree with class `SUV` and type `OFF` (i.e., `(SUV, OFF, {})`);
- Check the assignment, associating the previous typestate tree with the variable `suv`, and marking the expression on the right as aliased;
- Check the call `suv.turnOn()`, allowed in type `OFF`, generating “conditional” type information: if `true`, `suv` has typestate tree `(SUV, COMF_ON, {})`, otherwise it has `(SUV, OFF, {})`;
- Check the negating expression which “inverts” the conditional information;
- Inside the body of the `while` statement, `suv` is associated with `(SUV, OFF, {})`, and after exiting the `while`, `suv` is associated with `(SUV, COMF_ON, {})`;
- Check the call `suv.switchMode()`, which is allowed in type `COMF_ON`, generating “conditional” type information: `suv` has the typestate tree `(SUV, SPORT_ON, {})` if the call returns `Mode.SPORT`, and if the call returns `Mode.COMFORT`, it has `(SUV, COMF_ON, {})`. Since the returned value is not checked in a `switch` statement, we combine both typestate trees into `(SUV, SPORT_ON ∪ COMF_ON, {})`;
- Check the *parameter assignment* of `suv` by upcasting from `SUV` to `Car`, generating the typestate tree `(Car, ON, {(SUV, SPORT_ON ∪ COMF_ON, {})})`. Since the root type `ON` is a subtype of the required type in the `@Requires` annotation, the *parameter assignment* is allowed. Additionally, variable `suv` is marked as aliased: the `setSpeed` method is now the one responsible to complete the protocol of the given instance;
- No further checks are necessary for the call expression on `setSpeed` since it is a static method and methods are checked in a modular way;
- Type checking the `example` method finishes by checking protocol completion. Since all locations are marked as aliased at the end, no error about completion is reported.

To finish checking the class, we analyse `setSpeed`. The list of steps taken follows:

- We associate `car` with typestate tree `(Car, ON, {})`, according to the `@Requires` annotation;
- Downcast from `Car` to `SUV`, resulting in the typestate tree `(SUV, COMF_ON ∪ SPORT_ON, {})`;
- Check the call `((SUV)car).switchMode()`, which is allowed in type `COMF_ON ∪ SPORT_ON`, generating “conditional” type information: `(SUV)car` has typestate tree `(SUV, SPORT_ON, {})`, if the call returns `Mode.SPORT`, and if the call returns `Mode.COMFORT`, it has `(SUV, COMF_ON, {})`;
- To make `car` usable again, upcast to `Car`, associating `car` with the typestate tree `(Car, ON, (SUV, SPORT_ON, {}))`, if the call returned `Mode.SPORT`; and `(Car, ON, (SUV, COMF_ON, {}))` if the call returned `Mode.COMFORT`;

⁷ The repository of our tool includes an `examples` folder containing such examples.

- Check the `if` statement by propagating the type information corresponding to each branch;
- In the body of the `if` statement, downcast (again) from `Car` to `SUV`, resulting in the typestate tree (`SUV, SPORT_ON, {}`);
- Check the call `((SUV)car).setFourWheels(true)`, which is allowed in type `SPORT_ON`, in a similar fashion as before, associating `car` with `(Car, ON, (SUV, SPORT_ON, {}))`;
- Merge type information from both branches, resulting in `car` being associated with typestate tree `(Car, ON, (SUV, SPORT_ON ∪ COMF_ON, {}))`;
- Check the call `car.setSpeed(50)`, which is allowed in type `ON`, leading to `ON`;
- Check the call `car.turnOff()`, which is allowed in type `ON`, leading to `OFF`;
- Finish by checking protocol completion. Since all locations are marked as aliased or are in a final state (`car` is in the droppable typestate `OFF`), no completion error is reported.

Examples suite. We report the most significant examples of our suite in Table 1: **Directory** indicates the sub-directory; **Features** highlights the key features; **Checks** says if the example is accepted by our tool or not; and, **Runtime** describes the runtime error exhibited, if any.

In *Iterator (1)*, *Alarms* and *Cars (1)*, the examples test how our approach behaves with polymorphic code: as expected, the code compiles and no errors are thrown.

In *Drones (1)* and *Robots (1)*, the examples are more complex: we introduce a typestate data structure to increase the degree of flexibility (storing an arbitrary number of typestate objects, i.e., Drones and Robots). A key feature showcased here is the interaction between typestate objects: every time an object is used, it needs to be extracted from the data structure and put back once it has finished its task. In *Drones (2)*, the interaction between the data structure and the objects is even more articulate: we do not wait for the object to finish its task, but we immediately put it in the data structure and move to the next one, simulating a parallel tasks execution. The *Drones (3)* example is similar to the previous one, but it relies in a test for `null` being incorrectly negated in the return expression of an instance method, which causes a null pointer error in subsequent calls. The tool correctly propagates type information in the order methods may be called and catches this problem.

In *Iterator (2)* and *Cars (2)*, we show two problematic scenarios: index out of bounds and null-pointer exceptions, respectively. The former is caused by getting the next element without checking whether there are remaining elements or not. The latter is caused due to a field usage before initialising it. We are able to statically catch both cases.

Finally, in *Robots (2)*, we have another example of null-pointer error. The exception now is caused by a field being assigned to `null` in the subclass and used in the superclass, after performing an upcast. Thanks to our work, we are able to detect that, after assigning the field to `null`, the object is in a typestate with no supertypes, thus we raise an error.

In short, the provably sound theory presented in this paper is expressive and applicable to a mainstream object-oriented language, dealing with realistic code.

8 Related work

Fugue [13] allows checking typestates (seen as predicates over fields) by annotating methods with contracts and checking invariants. It handles casting and subclassing, where subclasses are allowed to introduce additional states with respect to superclasses. If an object ends up in a state unknown to its supertype, Fugue prohibits upcasting - as in our approach. To handle inheritance, *frame typestates* are introduced. Each frame is a set of fields declared in a particular class. An *object typestate* is the collection of frames. In our approach, our protocols

 **Table 1** Summary of examples.

Name	Directory	Features	Checks	Runtime
Iterator (1)	removable-iterator	Polymorphic safe code	Y	Ok
Iterator (2)	removable-iterator2	Wrong method call order	N	Out Of Bounds
Alarms	alarm-example	Polymorphic safe code	Y	Ok
Cars (1)	car-example	Polymorphic safe code	Y	Ok
Cars (2)	car-example2	Wrong method call order	N	Null-pointer
Drones (1)	drone-example	Typestated data structure	Y	Ok
Drones (2)	drone-example2	Typestated data structure Complex objects interaction	Y	Ok
Drones (3)	drone-example3	Same as Drones (2) and Incorrect test for <code>null</code>	N	Null-pointer
Robots (1)	robot-example	Typestated data structure Simple objects interaction	Y	Ok
Robots (2)	robot-example2	Wrong typestate upcast	N	Null-pointer

are globally defined with automata (e.g., List. 1 and 2), instead of method contracts, which we believe is more natural. Moreover, instead of using frames, we treat each class as a whole. This is enough since we view typestates as defining sequences of calls, not as predicates over fields, which simplifies the approach when dealing with overriding and dynamic dispatch.

Plural [8] statically checks that clients follow usage protocols based on typestates. It is based on earlier work [7] addressing the problem of substitutability of subtypes, while guaranteeing *behavioural subtyping* in an object-oriented language. Subtyping is supported by the programmer explicitly specifying which states “refine” (i.e., are substates of) others in the superclass. In our approach, we do not need to explicitly define subtyping relations: we define protocols in terms of state machines and automatically find all subtyping pairs.

Obsidian [12] is a language for smart contracts with a type system to statically detect bugs. It uses typestates to check state changes and has a permissions system for safe aliasing. It supports parametric polymorphism, but not casting to preserve strong static guarantees.

Gay et al. [18] extend earlier work on session types for object-oriented languages by attaching a protocol in the form of a session type to a class definition, and presenting an unification of communication channels and their session types, distributed object-oriented programming, and a form of typestates supporting non-uniform objects. The formal language includes a subtyping relation on session types [17] but does not include class inheritance (subtyping is just for channel communication). This approach has two implementations: Papaya [23] and Mungo [24]. Papaya considers protocols as in Gay et al. [18], but uses Scala as the target language with the same limitation of not coping with inheritance. Mungo considers protocols along the lines of Gay et al. [18], but uses Java (as we do) as the target object-oriented language. Inheritance is not supported apart from classes without protocols.

Bravetti et al. [11] present a type system for a Java-like language, where objects are annotated with usages, typestate-like specifications stating the allowed sequences of method calls. The type-based analysis ensures protocol compliance and completion, and memory safety (no null pointer dereferencing). However, subtyping (hence casting) is not supported.

Bouma et al. [10] develop a tool called BGJ that takes a global type, modelling the behaviour of processes in a multiparty session typing setting [20], and automatically generates Java classes modelling the APIs of projected local types. The state is encoded with a `state` field and transitions are encoded with methods annotated with preconditions and post-

conditions. To verify the clients of these APIs, the programmer writes Java code annotated with logical formulas. All annotations are statically checked by VerCors [9]. In our approach, one does not need to spread annotations throughout the code to specify or use protocols, we simply associate them with classes and the type system ensures memory-safety, protocol compliance and completion (properties the developer would need to specify for each program).

■ **Table 2** Comparison of related work.

Work	How protocols are defined	Casting approach
Fugue	Typestates are seen as predicates over fields and methods annotated with contracts	Handles casting with frame typestates
Plural	States defined as “refinements” of superclass states and methods annotated with contracts	Explicit specification of sub-typing relations
Obsidian	States defined explicitly and methods annotated with contracts	Casting disallowed for strong safety guarantees
Papaya	Usage types (i.e., automata-like)	Not supported
Mungo	Usage types (i.e., automata-like)	Not supported
BGJ	Scribble notation [28] projected to local types implemented as Java classes with <code>state</code> fields	Not supported
<i>JaTyC (ours)</i>	Usage types (i.e., automata-like)	Fully supported

9 Conclusions and future work

We overcome one of the main obstacles to the adoption of typestates in static analysis of object-oriented programs – the inability of performing cast operations freely at any point of the protocol – by introducing a novel theory based on typestate trees. We equip the theory with a set of functions to manage the typestate tree abstraction, and we mechanise soundness in the Coq proof system. We argue that typestate trees can be applied in various program analysers for object-oriented languages with inheritance, being thus language agnostic, opening the door for acceptance of several programs and features that were rejected until now in this kind of language. To support this claim, we implement a type checker for Java and assess the expressiveness of our approach. The relevance of the theory and of its applications is showcased by typestate-checking realistic Java code of an automotive system with driving dynamics control that allows to customise the drive mode of SUVs.

As future work, we plan to formally establish the runtime soundness of typestate trees by devising a core object-calculus with inheritance, static typestate semantics, and dynamic operational semantics, and by mechanising a type safety result: well-typed programs at runtime comply objects’ protocol with respect to both the order of method calls and its completion, and do not raise null-pointer exceptions. Additionally, we will study how these concepts can be adapted to a setting with multi-inheritance and generics.

References

- 1 Frances E Allen. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19, 1970. doi:[10.1145/390013.808479](https://doi.org/10.1145/390013.808479).
- 2 Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniéou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral Types in Programming Languages. *Found. Trends Program. Lang.*, 3(2-3):95–230, 2016. doi:[10.1561/2500000031](https://doi.org/10.1561/2500000031).

- 3 Lorenzo Bacchiani, Mario Bravetti, Marco Giunti, João Mota, and António Ravara. A Java typestate checker supporting inheritance. *Science of Computer Programming*, 221:102844, 2022. doi:[10.1016/j.scico.2022.102844](https://doi.org/10.1016/j.scico.2022.102844).
- 4 Lorenzo Bacchiani, Mario Bravetti, Julien Lange, and Gianluigi Zavattaro. A Session Subtyping Tool. In *Proc. of Coordination Models and Languages (COORDINATION)*, volume 12717 of *Lecture Notes in Computer Science*, pages 90–105. Springer, 2021. doi:[10.1007/978-3-030-78142-2_6](https://doi.org/10.1007/978-3-030-78142-2_6).
- 5 Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Intersection and Union Types: Syntax and Semantics. *Information and Computation*, 119:202–230, 1995.
- 6 Nels E Beckman, Duri Kim, and Jonathan Aldrich. An Empirical Study of Object Protocols in the Wild. In *Proc. of European Conference on Object-Oriented Programming (ECOOP)*, pages 2–26. Springer, 2011. doi:[10.1007/978-3-642-22655-7_2](https://doi.org/10.1007/978-3-642-22655-7_2).
- 7 Kevin Bierhoff and Jonathan Aldrich. Lightweight object specification with typestates. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005*, pages 217–226. ACM, 2005. doi:[10.1145/1081706.1081741](https://doi.org/10.1145/1081706.1081741).
- 8 Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007*, pages 301–320. ACM, 2007. doi:[10.1145/1297027.1297050](https://doi.org/10.1145/1297027.1297050).
- 9 Stefan Blom and Marieke Huisman. The VerCors Tool for Verification of Concurrent Programs. In Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods - 19th International Symposium. Proceedings*, volume 8442 of *Lecture Notes in Computer Science*, pages 127–131. Springer, 2014. doi:[10.1007/978-3-319-06410-9_9](https://doi.org/10.1007/978-3-319-06410-9_9).
- 10 Jelle Bouma, Stijn de Gouw, and Sung-Shik Jongmans. Multiparty Session Typing in Java, Deductively. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Proceedings, Part II*, volume 13994 of *Lecture Notes in Computer Science*, pages 19–27. Springer, 2023. doi:[10.1007/978-3-031-30820-8_3](https://doi.org/10.1007/978-3-031-30820-8_3).
- 11 Mario Bravetti, Adrian Francalanza, Iaroslav Golovanov, Hans Hüttel, Mathias Jakobsen, Mikkel Kettunen, and António Ravara. Behavioural Types for Memory and Method Safety in a Core Object-Oriented Language. In *Asian Symposium on Programming Languages and Systems*, volume 12470 of *Lecture Notes in Computer Science*, pages 105–124. Springer, 2020. doi:[10.1007/978-3-030-64437-6_6](https://doi.org/10.1007/978-3-030-64437-6_6).
- 12 Michael J. Coblenz, Reed Oei, Tyler Etzel, Paulette Koronkevich, Miles Baker, Yannick Bloem, Brad A. Myers, Joshua Sunshine, and Jonathan Aldrich. Obsidian: Typestate and Assets for Safer Blockchain Programming. *ACM Trans. Program. Lang. Syst.*, 42(3):14:1–14:82, 2020. doi:[10.1145/3417516](https://doi.org/10.1145/3417516).
- 13 Robert DeLine and Manuel Fähndrich. Typestates for Objects. In Martin Odersky, editor, *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Proceedings*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer, 2004. doi:[10.1007/978-3-540-24851-4_21](https://doi.org/10.1007/978-3-540-24851-4_21).
- 14 Edsger W. Dijkstra. The humble programmer, 1972. ACM Turing Award acceptance speech. doi:[10.1145/355604.361591](https://doi.org/10.1145/355604.361591).
- 15 Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. Foundations of Typestate-Oriented Programming. *ACM Transactions on Programming Languages and Systems*, 36(4):12, 2014. doi:[10.1145/2629609](https://doi.org/10.1145/2629609).
- 16 Simon J. Gay and Malcolm Hole. Types and Subtypes for Client-Server Interactions. In *Proc. of Programming Languages and Systems (ESOP)*, volume 1576 of *Lecture Notes in Computer Science*, pages 74–90. Springer, 1999. doi:[10.1007/3-540-49099-X_6](https://doi.org/10.1007/3-540-49099-X_6).

- 17 Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi-calculus. *Acta Informatica*, 42(2-3):191–225, 2005. doi:10.1007/s00236-005-0177-z.
- 18 Simon J. Gay, Vasco Thudichum Vasconcelos, António Ravara, Nils Gesbert, and Alexandre Z. Caldeira. Modular session types for distributed object-oriented programming. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010*, pages 299–312. ACM, 2010. doi:10.1145/1706299.1706335.
- 19 Tony Hoare. Null References: The Billion Dollar Mistake, 2009. Presentation at QCon London. URL: <https://tinyurl.com/eyipowm4>.
- 20 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Type. *J. ACM*, 63(1):9:1–9:67, 2016. doi:10.1145/2827695.
- 21 Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.
- 22 Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016. doi:10.1145/2873052.
- 23 Mathias Jakobsen, Alice Ravier, and Ornella Dardha. Papaya: Global Typestate Analysis of Aliased Objects. In *Proceedings of the 23rd International Symposium on Principles and Practice of Declarative Programming (PPDP’21)*, pages 19:1–19:13. ACM, 2021. doi:10.1145/3479394.3479414.
- 24 Dimitrios Kouzapas, Ornella Dardha, Roly Perera, and Simon J Gay. Typechecking protocols with Mungo and StMungo. In *Proc. of Principles and Practice of Declarative Programming (PPDP)*, pages 146–159. ACM, 2016. doi:10.1145/2967973.2968595.
- 25 Luis Mastrangelo, Matthias Hauswirth, and Nathaniel Nystrom. Casting about in the dark: an empirical study of cast operations in Java programs. *Proc. ACM Program. Lang.*, 3(OOPSLA):158:1–158:31, 2019. doi:10.1145/3360584.
- 26 João Mota, Marco Giunti, and António Ravara. On Using VeriFast, VerCors, Plural, and KeY to Check Object Usage (Experience Paper). In *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17–21, 2023, Seattle, Washington, United States*, volume 263 of *LIPICS*, pages 40:1–40:29. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.ECOOP.2023.40.
- 27 Fabian Muehlboeck and Ross Tate. Empowering union and intersection types with integrated subtyping. *Proc. ACM Program. Lang.*, 2(OOPSLA):112:1–112:29, 2018. doi:10.1145/3276482.
- 28 Rumyana Neykova and Nobuko Yoshida. Featherweight Scribble. In Michele Boreale, Flavio Corradini, Michele Loreti, and Rosario Pugliese, editors, *Models, Languages, and Tools for Concurrent and Distributed Programming - Essays Dedicated to Rocco De Nicola on the Occasion of His 65th Birthday*, volume 11665 of *Lecture Notes in Computer Science*, pages 236–259. Springer, 2019. doi:10.1007/978-3-030-21485-2_14.
- 29 Jens Palsberg and Pavlopoulou Chirstina. From Polyvariant flow information to intersection and union types. *Journal of Functional Programming*, 11(3):263–317, 2001. doi:10.1017/S095679680100394X.
- 30 Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D Ernst. Practical pluggable types for Java. In *Proc. of Software Testing and Analysis (ISSTA)*, pages 201–212. ACM, 2008. doi:10.1145/1390630.1390656.
- 31 R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, SE-12(1):157–171, 1986. doi:10.1109/TSE.1986.6312929.
- 32 Vasco T. Vasconcelos. Sessions, from Types to Programming Languages. *Bull. EATCS*, 103:53–73, 2011. URL: <http://eatcs.org/beatcs/index.php/beatcs/article/view/136>.

A Research Methodology

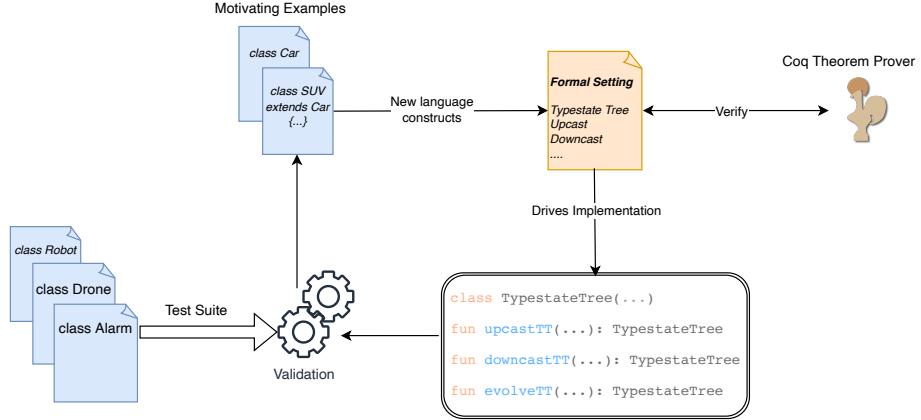


Figure 2 Research methodology behind the behavioural analysis support.

The iterative process in Figure 2 shows our methodology to support behavioural analysis within our type checker: we extract, from motivating examples, the language features to include in the static analysis; we build a formal setting (verified in the Coq theorem prover) to drive the JaTyC implementation; finally, we validate our approach with a suite of examples.

B Glossary

- m A meta-variable ranging over the set of method identifiers **MNames**
- o A meta-variable ranging over the set of output values **ONames**
- s A meta-variable ranging over the set of typestate names **SNames**
- \tilde{A} The wide tilde stands for a sequence of values
- $d\{\tilde{m} : \tilde{w}\}$ An input state (Definition 1)
- $\langle \tilde{o} : \tilde{u} \rangle$ An output state (Definition 1)
- w A meta-variable ranging over input and output states, and typestate names (Definition 1)
- u A meta-variable ranging over input states and typestate names (Definition 1)
- \tilde{E} A set of defining equations
- $w^{\tilde{E}}, u^{\tilde{E}}$ A meta-variable to denote a state w (resp. u) with a set of defining equations
- \mathcal{W}, \mathcal{U} The set of terms $w^{\tilde{E}}$ (resp. $u^{\tilde{E}}$)
- \mathcal{X} A subset of $w^{\tilde{E}}$ containing only input states $d\{\tilde{m} : \tilde{w}\}$
- \mathcal{Y} A subset of $w^{\tilde{E}}$ containing only output states $\langle \tilde{o} : \tilde{u} \rangle$
- \leq_S A subtyping relation between states (Definition 4)
- $\leq_{S_{alg}}$ The algorithmic version of the subtyping relation between states (Definition 7)
- t A meta-variable ranging over the set of types \mathcal{T} (Definition 10)
- \leq A subtyping relation between types (Definition 11)
- c A meta-variable ranging over the set of class names \mathcal{C}
- \leq_C A subtyping relation between classes (Definition 15)
- Ret_m The set of outputs returnable by a method m
- tt A meta-variable ranging over the set of typestate trees \mathcal{TT} (Definition 36)
- tts A meta-variable ranging over $\mathcal{P}(\mathcal{TT})$
- \vdash Well-formedness of typestate trees (Definition 38)
- $\vdash_{c,t}$ Soundness of typestate trees (Definition 53)

Cross Module Quicken – The Curious Case of C Extensions

Felix Berlakovich 

University of the Bundeswehr Munich, Neubiberg, Germany

Stefan Brunthaler 

University of the Bundeswehr Munich, Neubiberg, Germany

Abstract

Dynamic programming languages such as Python offer expressive power and programmer productivity at the expense of performance. Although the topic of optimizing Python has received considerable attention over the years, a key obstacle remains elusive: C extensions. Time and again, optimized run-time environments, such as JIT compilers and optimizing interpreters, fall short of optimizing *across* C extensions, as they cannot reason about the native code hiding underneath.

To bridge this gap, we present an analysis of C extensions for Python. The analysis data indicates that C extensions come in different varieties. One such variety is to merely speed up a single thing, such as reading a file and processing it directly in C. Another variety offers broad access through an API, resulting in a domain-specific language realized by function calls.

While the former variety of C extensions offer little optimization potential for optimizing run-times, we find that the latter variety *does* offer considerable optimization potential. This optimization potential rests on *dynamic locality* that C extensions cannot readily tap. We introduce a new, interpreter-based optimization leveraging this untapped optimization potential called Cross-Module Quicken. The key idea is that C extensions can use an optimization interface to register highly-optimized operations on C extension-specific datatypes. A quickening interpreter uses these information to continuously specialize programs with C extensions.

To quantify the attainable performance potential of going beyond C extensions, we demonstrate a concrete instantiation of Cross-Module Quicken for the CPython interpreter and the popular NumPy C extension. We evaluate our implementation with the NPBench benchmark suite and report performance improvements by a factor of up to 2.84.

2012 ACM Subject Classification Software and its engineering → Runtime environments; Software and its engineering → Interpreters

Keywords and phrases interpreter, optimizations, C extensions, Python

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.6

Supplementary Material

Software (Docker image for Artifact Evaluation): <https://doi.org/10.5281/zenodo.11174717> [6]

Software (WIP code of the NumPy part of CMQ): <https://github.com/fberlakovich/cmq-numpy-ae> [5]

Software (Updated code of the CPython part of CMQ): <https://github.com/fberlakovich/cmq-ae> [4]

Funding The research reported in this paper has been funded by the Federal Ministry for Climate Action, Environment, Energy, Mobility, Innovation and Technology (BMK), the Federal Ministry for Labour and Economy (BMAW), and the State of Upper Austria in the frame of the COMET Module Dependable Production Environments with Software Security (DEPS) [FFG grant no. 888338] and the SCCH competence center INTEGRATE [FFG grant no. 892418] within the COMET – Competence Centers for Excellent Technologies Programme managed by Austrian Research Promotion Agency FFG.

 © Felix Berlakovich and Stefan Brunthaler;
licensed under Creative Commons License CC-BY 4.0

38th European Conference on Object-Oriented Programming (ECOOP 2024).

Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 6; pp. 6:1–6:29

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

1 Motivation

Productivity or performance? Despite the ever-increasing performance of computers, software developers are faced with this conundrum. They can either choose a high-level language like Python to benefit from abstractions like dynamic typing or garbage collection, but sacrifice performance. Alternatively, they can resort to low-level programming languages like C or C++ to gain better performance, but at the cost of developer productivity and safety.

According to the TIOBE index, the popularity of high-level languages such as Python or Ruby is unbroken¹ [36]. At the same time, however, the poor performance of these high-level languages remains an ongoing problem for, e.g., Python or Ruby [2, 39, 31]. Recent efforts address the performance issues of Python and Ruby [16, 38, 37, 40, 10, 9, 11, 12, 15, 14, 34, 35].

Besides the language VMs themselves, Ruby and Python, also have a thriving ecosystem of C extensions. C extensions, however, do not profit from optimizing the language VM. With the ongoing VM optimization efforts and the ensuing increase in performance of the core language, the performance of C extensions could come into focus in the near future.

C extensions also pose an optimization barrier for JIT compilers like PyPy or YJIT [17]. Due to the lack of semantics, JIT compilers cannot reason across the boundary of the core language. As a result, JIT compilers *cannot fully optimize* at the interface to C extensions or even into the extension code. A common workaround is to reimplement the entire extension in the host language (e.g., Python), thus removing the lack of semantics and closing the gap between VM and extension. For example, the PyPy project includes a pure Python implementation of a subset of NumPy to enable more aggressive optimizations. This approach has improved performance substantially in some cases, but requires a full or at least partial rewrite of the extension.

The two approaches of (i) not optimizing extensions at all, or (ii) rewriting them in the host language to make them accessible for JIT compilers, occupy two extremes on the design spectrum. In this paper, we explore an additional way of optimizing the interaction of high-level language code with C extensions.²

We first provide a short analysis of the different C extension varieties, based on popular² C extensions for Python (see Section 3). Our analysis indicates that some C extensions focus on a single, isolated task, which is implemented in optimized C. This variety does not lend itself well to optimization and would also not profit from JIT compilation in many cases. The other variety provides a broader API and custom datatypes, effectively exposing a domain-specific language through an API. This second variety offers a larger optimization potential.

To tap this potential, we introduce a new, interpreter-based optimization technique called *Cross-Module Quickening*, or CMQ for short (see Section 4). CMQ allows the interpreter, in collaboration with the C extension, to extend the interpreter’s optimization effort *into* the extension. The key idea is to provide the C extension with an interface to register specialized, extension-specific interpreter instructions. These specialized derivatives allow extension authors to exploit, for example, type locality within the C extension that would otherwise be invisible to the interpreter. Our technique does not require any changes, such as type annotations, in the Python program. CMQ also does not depend on runtime code-generation and is, thus, suitable for resource-constrained devices.

To demonstrate our idea, we analyze the optimization potential in NumPy, a popular Python C extension (see Section 5.2.5 and Section 6). We provide specialized derivatives for a number of NumPy operations and achieve a speedup of up to 2.84x in NPBench, a collection of compute-intensive NumPy programs (see Section 7).

¹ Python, for example, has continuously gained in popularity since 2018 and even leads the trends for 2024, so far

² The PyPI statistics range back only one month.

Summing up, this paper contributes the following

- We present Cross-Module Quicken, or CMQ for short, a new interpreter-based optimization architecture to optimize *across* C extensions. CMQ introduces a so-called Optimization Interface that allows C extensions to provide optimized instructions, thereby enabling cross-module type feedback via inline caching.
- We classify different use cases of C extensions with respect to their performance potential. We find that it is presently impossible to conduct an extensive quantitative analysis. The key obstacle is due to each C extension requiring varying amounts of domain expertise, usually provided by a human that has experience in using a given C extension. To shed light into the C extension “black box,” we conduct a qualitative analysis on the top ten C extensions instead.
- We describe the relevant details of a concrete CMQ implementation for the CPython interpreter and the NumPy extension. This concrete implementation introduces novel interpreter optimization techniques, such as extension-delimited superinstructions, and per-instruction caches for C extensions.
- We report the results of a comprehensive evaluation that encompasses the following dimensions: dynamic locality, performance, and implementation effort. Specifically, an in-depth analysis of NPBench on NumPy finds:
 - Quantitative dynamic locality of about 99%.
 - Performance improvements by a factor of up to 2.84.
 - Moderate implementation effort of less than 4,000 lines of code in CPython and NumPy.

2 Background

2.1 C Extensions

Most language VMs offer a way to interact with native code, typically called *foreign function interface*. Several language VMs go one step further by allowing *native code extensions*. These extensions are not limited to merely providing functions that can be called from the host language via a foreign function interface. Instead, an extension can define arbitrary host language types and modules, and even manipulate the VMs runtime state through an API. *Extension* means that the language VM loads the code dynamically at *runtime*, as opposed to code that is integrated at build time (e.g., CPython’s `sqlite3` extension). For example, Python, Ruby, and Lua all offer such extension APIs.

In principle, native extensions can be written in *any* language that compiles to native code and can access the VM’s APIs. Since C is the most popular language for native extensions, however, we will refer to native extensions collectively as *C extensions* from now on. Nonetheless, the principles described in this paper apply to native extensions written in any language.

2.2 Type Feedback via Inline Caching

Inline caching, first introduced by Deutsch and Schiffmann in 1984, is a technique for optimizing dynamic languages [18]. The technique is particularly useful for language VMs featuring generic operations. Many language VMs, for example, have a generic `BINARY_ADD` operation that can add two operands with arbitrary types, such as integers or floats.

To deal with the different semantics of, e.g., adding integers compared to adding floats, the language VM needs to resolve the concrete implementation dynamically based on the operand types. Depending on the number of supported types and implementations, this lookup

process can be expensive. The important observation behind inline caching is that even for dynamically typed programs, the operand types for an operation hardly ever change, if at all. Deutsch and Schiffmann called this principle *dynamic locality of type usage* [1, 18, 40].

A language VM can leverage this locality and cache the result of the expensive lookup process. In the example of `BINARY_ADD` above, the language VM could cache a pointer to the concrete implementation of e.g., integer addition. Since this cache typically resides *inline* with the instructions, i.e., no additional redirection is needed to access the cache, it is called an *inline cache*. Next time the language VM encounters this particular occurrence of `BINARY_ADD`, it can use the cached pointer instead of resolving the concrete implementation again. Before using the cache, however, the language VM needs to check that the operand types are equal to the expected types. In the unlikely case that the operand types *have* changed, the runtime would invalidate the inline cache.

2.3 Quicken – Instruction Rewriting to Capture Runtime Knowledge

Another interpreter optimization technique is called *quicken*. Quicken describes a process where an interpreter uses runtime feedback, such as type usage, to rewrite generic instructions to more concrete ones. This principle was originally used for efficiently resolving `classpool` references in the Java virtual machine [29]. The more concrete instructions are sometimes called *optimized derivatives* or just *derivatives*.

An example is a generic `BINARY_OP` instruction, whose operation depends on its operand. `BINARY_OP` with argument 1 performs an addition, whereas with argument 2 it performs a subtraction. If the language VM observes that a particular `BINARY_OP` always performs a subtraction, it can rewrite the instruction to `BINARY_SUBTRACT`. A `BINARY_SUBTRACT` no longer has to consult its argument value, but can perform a subtraction directly.

Quicken is a way for the language VM to encode temporal locality in its instruction set. Depending on the observed information, the encoded state is either permanent or transient, but with a high likelihood. If the state is permanent, the quickened instruction does not need to check any assumptions. If it is only likely, however, the language VM needs to validate the assumptions under which the quickening occurred. If the program invalidates an assumption, the language VM needs to rewrite affected instructions back to their original, generic form. For example, if a quickened instruction depends on specific operand types, and the operand types change, the language VM, needs to revert the instruction to a type-generic instruction. As the language VM speculates on the stability of the observed information, this optimization is typically called *speculative optimization*. This reversal of an optimized instruction back to its original form typically called *deoptimization*.

2.4 Inline Caching and Quicken in Python

Our implementation of CMQ builds on top of CPython and its existing optimization infrastructure. To aid the understanding of our implementation, we give a short overview of the related techniques here. CPython uses a combination of inline caching and quickening. Specifically, CPython uses specialized instructions, some of which also have an inline cache. The instructions with inline cache, such as `LOAD_GLOBAL_MODULE`, store assumption-related data in the cache that allows them to deoptimize if any of the assumptions change. Other instructions, like `BINARY_ADD_INT`, validate the assumptions without an inline cache (e.g., by directly checking the operand types). That is, their assumptions are directly encoded in the instruction set [13].

Extension	Categories
brotli	binder
cryptography	binder
matplotlib	optimizer,binder
Pillow	optimizer
PyYAML	optimizer

Extension	Categories
Tensorflow	extender,optimizer
NumPy	extender
Pandas	extender
CuPy	extender
PyTorch	extender,optimizer

(a) Python extensions with little room for C extension optimization.

(b) Python extensions with custom datatypes, operator overloading and new surface syntax (extenders).

■ **Figure 1** Overview of the Python C extensions we considered for CMQ. The extensions on the left are binders and/or optimizers. The extensions on the right are extenders.

A peculiarity of CPython is that it uses the inline cache also to store profiling data that controls the quickening process. Specifically, instructions with specialized derivatives, store a counter in the inline cache. Every generic instruction derivative (e.g., LOAD_GLOBAL) decreases the counter upon execution. Once the counter reaches zero, the instruction tries to quicken itself to one of the specialized derivatives (e.g., LOAD_GLOBAL_MODULE). Thus, the counter implements a warmup phase in which the involved operands and types can stabilize. Likewise, when a specialized derivative has to deoptimize due to an invalidated assumption, it increases the counter by a certain *backoff* value. The backoff value ensures that an instruction with varying operand types does not continuously swap between two derivatives.

CPython organizes generic instructions and their specialized derivatives in *instruction families*. Each member of an instruction family has the same inline cache size. The inline cache is located directly after the instruction in the instruction stream. Each instruction is responsible for skipping the cache after instruction execution.

3 C extensions of Dynamic Languages

In this section we describe the results of our investigation of Python’s C extension ecosystem. Although we focus explicitly on Python, we believe that our findings generalize to similar ecosystems, such as Ruby or Lua.

3.1 Domain Specificity of C Extensions

Our initial plan was to conduct a large-scale, quantitative analysis of C extensions. After some experimentation and manual investigation, however, we found this goal to be elusive. This failure is due to C extensions being *domain specific*. They solve a single, well-defined problem, but do so in radically different ways. Ways that do not generalize from one C extension to another, and, therefore, pose a substantial obstacle to automation, the prerequisite for a large-scale analysis and quantitative investigation.

The domain specificity of C extensions not only frustrates generalized analysis attempts. Our performance analysis of C extensions identified a symmetric problem: If one lacks the domain expertise to tell what a “good use case” for a C extension is, it is nigh impossible to perform unbiased experiments.

These initial findings led us to conduct a qualitative analysis using manual investigation instead.

3.2 Of Optimizers, Binders, and Extenders

We analyzed the C extensions for Python in Figure 1 and found that they fall, broadly speaking, into three categories:

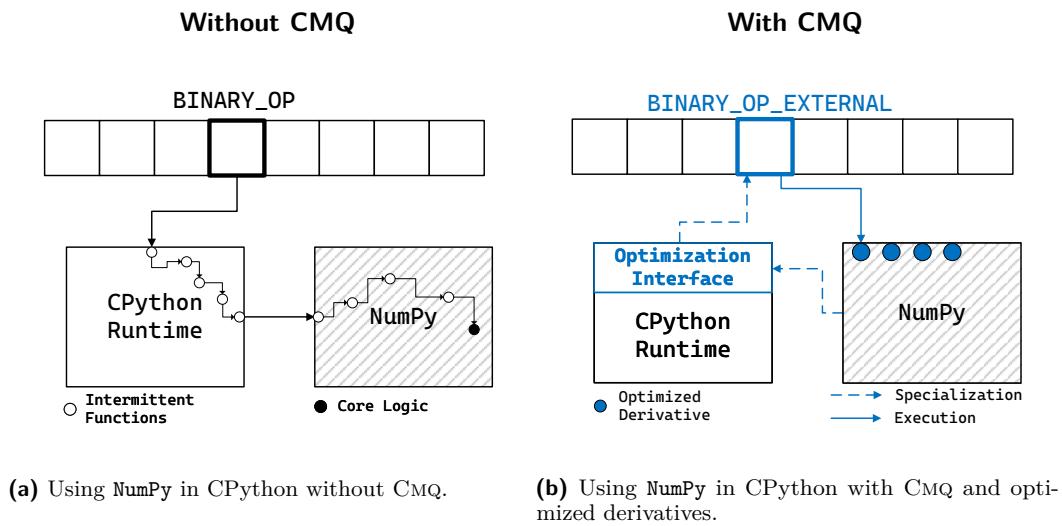
1. *Optimizers*: These C extensions could in principle be written in Python, and some of them probably were initially Python libraries. Due to the proverbial need for speed, however, these libraries are written in C, thereby eliminating a lot of the performance overhead associated with Python. Often, these C extensions offer just a single point of entry, execute efficiently in machine code, and return Python-processable data.
Consider the PyYAML extension as an example: The interface is just one call to the `parse` routine, which performs all the parsing in C, and returns the corresponding configuration data.
2. *Binders*: These C extensions usually cannot be written in Python, because they provide bindings to existing libraries to the Python ecosystem. These libraries are written in another language, such as C and C++, and bindings are the intermediary layer that translates from one world to another. The functionality corresponds to the external library, or a subset thereof that is reasonable to use from within Python.
Consider the lxml extension as an example: The interface corresponds to the libxml library, which implements efficient, feature-rich, and standards-compliant XML parsing.
3. *Extenders*: These C extensions extend Python with functionality not readily present in Python itself. These extensions define custom datatypes, overload and/or misuse operators, and at times resort to a custom embedded-DSL modeled through function calls. Note that extensions in this category are not mutually exclusive to others, as they can also embed existing libraries into their functionality.

Consider the NumPy extension as an example: NumPy defines its own datatype, a multi-dimensional array mapped to contiguous memory . This feature extends Python, as in Python a list or an array behaves similar to Java jagged arrays, i.e., each dimension is just a single array, which maps to another dimension, being a single dimensional array again. In contrast to Python lists, NumPy’s array representation enables high-performance operations on these arrays.

Through the performance optimization lens, the first two categories offer little potential for performance optimization. This lack of potential is due to their inner workings. PyYAML, for example, slurps a YAML file into C, parses the file efficiently using native-machine code, and creates the corresponding Python objects. Since this has been highly optimized already, no optimization opportunity presents itself. Similarly, lxml is just a small layer that invokes libxml to do the heavy lifting. No complex and expensive processing is done within the C extension. Both of these effects are amplified further by the old adage that time spent in libraries is lost w.r.t. optimization [20].

3.3 Exploring Extenders

Extenders, i.e., C extensions enriching the Python programming language do so in various ways. These extensions provide custom data types, such as NumPy providing a multi-dimensional array that is mapped to contiguous memory. Since our target languages are dynamically typed, manipulation of custom data types relies upon *operator overloading*. On top of these data types, C extensions have the possibility to (ab-)use existing functionality to introduce *surface syntax*. NumPy, for example, (ab-)uses Python’s tuples to provide a way to encode multi-dimensional array index access. Where no such surface syntax is available, Extender C extensions resort to using function calls. In combination these properties form a type of embedded DSL.



■ **Figure 2** Without CMQ (left), C extension-calls need to go through a cascade of function calls before reaching the core logic. With CMQ (right), the language VM calls optimized derivatives directly.

In contrast to Optimizers and Binders, programs using Extenders frequently cross the boundary between language VM and C extension. Context such as type locality established by the language VM or the C extension does not cross this boundary, leading to redundant checks and missed optimization potential. In Section 4 we discuss how CMQ lifts this optimization potential. To give concrete examples, we will now focus on the NumPy C extension, which adds high-performance numeric processing to Python.

3.4 Summary of Observations

Let us briefly summarize our findings, which are of vital importance for the following Sections.

- C extensions require domain expertise to analyze and evaluate.
- Only one of three categories offers dormant optimization potential.
- The Extenders category of C extensions form a kind of embedded DSL, by providing custom types, operator overloading, or introducing surface syntax.

4 Design of Cross-Module Quickening

The goal of CMQ is to enable optimizations across extension boundaries. Figure 2 gives an overview of CMQ. Without CMQ (Figure 2a), each operation involving a C extension must go through a cascade of function calls. At present, the interface between language VM and C extension poses an optimization boundary. As a result, the function calls are necessary to reestablish context that was already established previously, or on the other side of the optimization boundary (language VM vs C extension).

To eliminate this overhead, CMQ proceeds as follows (see Figure 2b):

1. CMQ provides a dedicated *Optimization Interface* or OINT for short, which enables C extensions to provide domain-specific optimizations.
2. Based on context information, the C extension can use quickening-based optimization through optimized interpreter instructions.

3. The interpreter provides an interface to replace single generic instructions or entire instruction sequences with optimized ones.
4. Optimized instructions validate that their assumptions hold and deoptimize upon misspeculation.
5. Additional optimization opportunities for C extensions exist, for example, through having per-instruction caches.

The following sections explain the relevant conceptual design details with examples from `CPython` and `NumPy`. Each section also contains forward references to the relevant implementation details in CMQ. Although we discuss implementation details primarily for the `NumPy` C extension, the principles underlying this specific implementation generalize not only to other C extensions, but also to C extension ecosystems of other dynamic programming languages. For brevity, we call our modified `NumPy` CMQ-`NumPy`.

4.1 Optimization Interface

C extensions for language VMs such as Ruby or Python are implemented as dynamically loadable modules. This means that C extensions and the language VM communicate via a predefined interface. Typically, the interface consists of both, public APIs in the language VM and hooks in the C extension called by the language VM. For example, `CPython` automatically calls public `PyInit_*` functions exposed in a loaded C extension. These functions create module objects for each module provided by the C extension. At the same time, `CPython` exposes functions to e.g., query the type of objects or to create new objects such as dictionaries.

We extend this interface between language VM and C extensions with an optional Optimization Interface, or OINT for short. The goal of the OINT is to expand the interpreter’s optimization capabilities with domain-specific optimizations. To that end, the OINT allows a C extension to register an *instruction optimization hook*. One goal of the OINT is to shield the C extension from as many language VM specific implementation details as possible.

Whenever the language VM tries to optimize an instruction, it calls all registered instruction optimization hooks. When exactly an optimization attempt happens, depends on the concrete architecture of the language VM. For example, optimization can happen either as part of an instruction’s execution (as is common for quickening) or in a dedicated optimization phase (as is common in JIT compilation). `CPython` performs instruction quickening as part of the generic instruction’s execution, once an optimization counter reaches zero (see Section 2.4).

The exact contract of the instruction optimization hook depends on the concrete language VM implementation. In general, the language VM needs to provide the C extension with enough information to decide which optimizations are applicable. For example, in CMQ-`NumPy`, the instruction optimization hook receives a pointer to the current instruction and a pointer to the operand stack.

The optimization of an instruction through a C extension is *optional*. Based on the instruction and its operands, a C extension can decide which optimizations are applicable, if any. For example, the C extension can query the operand types to leverage dynamic-type locality. CMQ-`NumPy` uses this principle to optimize certain `BINARY_OP` occurrences. We give a more detailed description of the `BINARY_OP` optimization in Section 6.2. In addition to type checks, the C extension can inspect further properties of the operands to decide whether optimizations are applicable. In Section 6.2 we describe how CMQ-`NumPy` inspects the name of `NumPy` `ufunc` objects to decide whether it can optimize specific `CALL` instructions.

4.1.1 Validating Assumptions and Deoptimization

As discussed in Section 2.3, quickening optimizations can be speculative. To guarantee correctness, the language VM needs a way to detect invalid assumptions and restore the original instructions. One strategy of validating assumptions is as part of the optimized instruction’s execution. For example, our `BINARY_OP` derivatives verify that the operands on the stack have the expected types.

Performing the assumption validation in the operation itself works well for assumptions about operands, such as their types, but is less suited for assumptions concerning global properties. For example, in addition to specific operand types, our `BINARY_OP` derivatives assume NumPy’s default arithmetic implementations for e.g., adding and subtracting arrays. While a user *can* change the implementations by overriding fields in the NumPy module, it happens rarely. Similar to operand types, each optimized derivative could validate this assumption before execution. However, with such an implementation each derivative suffers from a small performance overhead to check for an event that occurs infrequently. To mitigate this cost, the OINT offers an alternative way to validate assumptions. Specifically, the OINT allows C extensions to record deoptimization triggers for optimized instructions. Any code within a C extension that modifies properties previously optimized derivatives depend on, needs to notify the OINT. The OINT then deoptimizes all affected optimized instructions. Code that changes any of NumPy’s default arithmetic implementations, for example, triggers an deoptimization event. In response, CMQ deoptimizes all `BINARY_OP` derivatives. This approach shifts the burden of assumption validation to the infrequent path of changing arithmetic implementations.

A hybrid between the previous two approaches of deoptimization is to combine multiple object properties into a *meta-property*. For example, our `CALL` derivatives optimize calls of NumPy `universal functions`, or `ufunc` for short. The `ufunc` object is a stack operand of the corresponding `CALL` derivative. In addition to validating the `ufunc` operand’s type, the `CALL` derivative needs to verify several additional properties. For example, the `CALL` derivative is only valid for `ufuncs` without custom user loops. Verifying all these properties individually causes a performance overhead and, thus, reduces the profit of the `CALL` derivative optimization. Instead, we change the `ufunc` object to maintain a meta-property in the form of a `specializable` flag. The `specializable` flag represents the state of all individual properties combined. Code that updates any of the individual properties, also updates the `specializable` flag accordingly. Instead of validating all `ufunc` properties individually, the `CALL` derivative now has to validate only the `specializable` flag. Figure 3a illustrates this process graphically. With this approach, the burden of assumption validation is *shared* between code that modifies `ufunc` properties and the `CALL` derivatives.

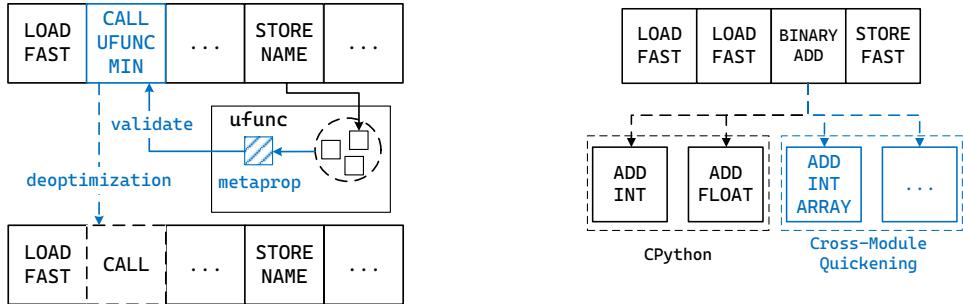
We describe our implementation of the specialization infrastructure for CPython in more detail in Section 5.2.5.

4.2 Cross-Module Optimization Opportunities

4.2.1 Type-specialized Instructions

In Section 2.2 and Section 2.3 we discussed the principle of *locality of type usage*. CPython leverages this principle to quicken type-generic instructions to type-dependent instructions. For example, CPython quickens `BINARY_OP` to `BINARY_OP_ADD_INT`, a derivative that directly adds the two integer operands. Compared to the generic instruction, the derivative’s call stack contains *three* fewer frames when reaching the final `_PyLong_Add` function. In addition, the derivative saves multiple intermediate calls needed to resolve the concrete function that adds Python integers. More specifically, the derivative saves the following steps performed by the generic instruction:

6:10 Cross Module Quickening – The Curious Case of C Extensions



(a) CMQ uses meta-properties to efficiently validate assumptions (see Section 4.1.1).

(b) CMQ allows to quicken instructions with domain-specific derivatives.

Figure 3 Overview of meta-properties (left) and type-specialized instructions (right) in CMQ.

1. Check if any of the operands has an implementation for the + slot.
2. Check if the left operand is a number and has the + slot.
3. Check if the right operand is a number of a different type than the left operand and has a different + slot.
4. Depending on whether the right operand has a different + slot and is a subtype of the left operand, call the left or right operand's + slot.
5. In the slot implementation, ensure that both operands are actually Python Longs.

List 1 Steps for resolving the implementation for adding two integers in CPython.

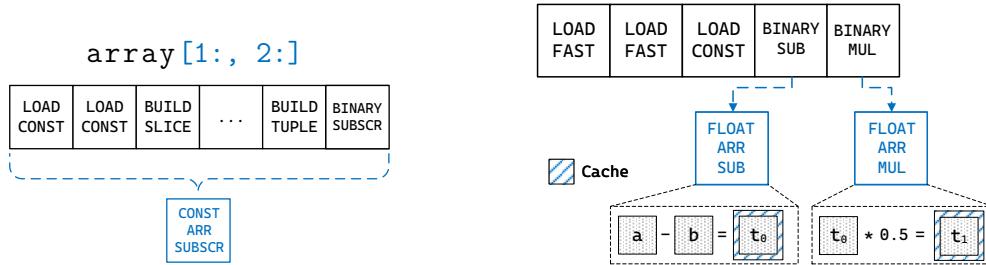
Under the assumption that both operands are Python Longs, the final operation (`_PyLong_Add`) is known immediately and the intermediate steps in List 1 become redundant. However, a language VM can only leverage locality of type usage, if it knows the types and operations involved. For example, the special handling of integer addition in CPython is only possible if both operands are non-subtyped Python Longs. If the language VM cannot reason about a type, such as a type provided by a C extension, quickening is no longer possible.

This issue is exacerbated by C extension types. Depending on the domain and the extension, the C extension has to check additional properties to the ones in List 1. We describe the additional checks that NumPy performs in more detail in Section 6.1. Similarly to the checks in List 1, the additional checks in NumPy are strongly connected to the operands' types. That is, under the assumption of specific operand types, the majority of checks become redundant.

Based on this observation, CMQ enables type-specialized instructions that depend on C extension types. CMQ-NumPy, for example, provides specialized instructions that add two double precision floating point arrays. As a result, starting from the interpreter loop, the call stack for adding two such arrays collapses from 13 frames to 2. In addition, the specialized instructions save several intermediate checks. These checks are subsumed by the fixed number and types of operands involved.

4.2.2 Extension-delimited Superinstructions

Replacing generic instructions with type-specialized instructions renders many of the checks performed by the generic instruction redundant (see Section 4.2.1). With the OINT, a C extension is not limited to replacing a single instruction at a time, however. In certain



(a) CMQ can replace instruction sequences, such as custom array subscripts, with a single optimized instruction (see Section 4.2.2).

(b) Caching data between instruction executions (see Sections 4.2.3 and 6.4).

■ **Figure 4** Illustration of extension-delimited superinstructions (left) and per-instruction caches (right).

cases, a specialized instruction subsumes the result of an entire sequence of instructions. For example, for `BINARY_SUBSCRIPT` instructions with constant indices, such as `array[1:, 2:]`, CMQ-NumPy precomputes the index structure during specialization. With the index structure computed, all the index operands become redundant. As a result, the instructions pushing these operands onto the operand stack are now *dead code* in program analysis terminology. To account for such cases, the OINT allows to replace *entire sequences* of instructions with specialized derivatives, as shown in Figure 4a.

By subsuming multiple unoptimized instructions, the optimized derivative represents a type of *superinstructions*. Unlike conventional superinstructions however, the boundaries of the superinstructions enabled by CMQ are domain-specific and defined by the C extension. Thus, we call this type of superinstruction *extension-delimited superinstruction*.

4.2.3 Caching Between Instruction Executions

Specialized instructions can efficiently encode type membership and similar properties with a low information density (see Section 4.2.1). For example, type membership is representable as a single bit in the instruction encoding. Some optimizations, however, depend on data that is hard to encode in an instruction, but instead need a dedicated cache. NumPy's arithmetic instructions, for example, frequently allocate new arrays, which are deallocated only a few instructions later. At the expense of a little additional memory, optimized derivatives can keep a cached result array to avoid repeated allocations and deallocations. To that end, the OINT provides a mechanism to store data in a cache space specific to an instruction *occurrence*. We call this cache *occurrence cache*. Conceptually, the *occurrence cache* allows an instruction to communicate data between instruction executions or between specialization time and execution. We describe instantiations of both variants in more detail in Section 6.4.

The *occurrence cache* acts like an inline cache, but it is implementation-specific. To the C extension it is opaque whether the language VM actually stores the cache inline. Also, in contrast to the typical usage of an inline cache, i.e., storing function pointers, the *occurrence cache* can store arbitrary data, including data pointers. We describe our implementation of the cache space in more detail in Section 5.2.3 and how we use the cache in Section 6.4.

5 Implementation of Cross-Module Quickening in CPython

In this section, we start with a short overview of CPython’s internal implementation and then describe the integration with CMQ.

5.1 CPython in a Nutshell

The CPython interpreter is a stack machine with instructions that consist of an `opcode` and an `oparg`. The `opcode` specifies what an instruction does and is one byte long. The `oparg` serves different purposes, depending on the instruction, and is also one byte long. For example, in the `LOAD_FAST` instruction, the `oparg` specifies which local-variable slot to push onto the operand stack.

Instructions with inline caches are grouped into families. All members of the same family are specializations of a generic instruction that is also part of the family. Family members have an equally sized inline cache (see Section 2.4 for more details).

The snippets of code that implement an instruction’s semantics are called *opcode handler*. CPython uses *indirect threading*, which means that each `opcode` handler jumps to the next handler through a dispatch table [21]. A compiler feature called `computed gotos` allows an efficient compilation of such dispatch patterns.

5.2 Integration with Cross-Module Quickening

CMQ enables C extensions to replace generic interpreter instructions with optimized derivatives. When integrating CMQ with CPython’s dispatch routine, we faced a number of competing constraints:

1. Specialization should happen as soon as possible to unlock additional performance. However, the language VM should not repeatedly try to specialize an instruction if no specialization is possible or if the instruction deoptimized recently.
2. Considering that specialization happens for *hot code*, the execution of external, optimized derivatives should be as fast as possible.
3. CMQ needs to avoid consuming too much of CPython’s already limited `opcode` space.
4. CPython can load multiple C extensions simultaneously, each of which could potentially register optimized derivatives. In addition, each C extension can register multiple different derivatives for the same generic instruction. Therefore, CMQ must allow the registration of as many derivatives as possible.
5. While specialization and deoptimization happens infrequently compared to an instruction’s execution, the time spent on these tasks must eventually be amortized. Thus, specialization and deoptimization must be reasonably fast, or they defeat the purpose of optimization.

In the following subsections, we describe our design choices for CMQ and how each decision relates to the aforementioned challenges.

5.2.1 Specializing Hot Instructions

For CMQ, we extend CPython’s existing quickening mechanism to consider not only CPython derivatives, but to also call registered instruction optimization hooks (if any). Extending the existing mechanism allows CMQ to leverage CPython’s optimization counter infrastructure. The optimization counter ensures that CMQ (1) only attempts to specialize hot instructions and (2) that each failed optimization attempt delays further attempts by an increasing value. Specifically, if both, CPython’s internal optimizations and the optimization function, fail to optimize an instruction, CPython increases the optimization counter by a backoff value (see Section 2.4).

5.2.2 External opcode handlers

Ideally, C extensions could register `opcode` handlers that resemble internal `opcode` handlers. **Computed gotos**, however, are only possible within a single function. The C standard considers jumps into the middle of a function from outside the function undefined behavior. In **CPython**, therefore, an exact resemblance of internal `opcode` handlers is not possible. Instead, we resort to subroutine-threading for the external `opcode` handlers, i.e., we implement each handler as a function in the C extension.

5.2.3 Dealing with a Limited Opcode Space

CPython's small `opcode` encoding of one byte means that few opcodes remain for specialization through C extensions. Specifically, **CPython** 3.12 has 208 opcodes, leaving 47 opcodes undefined. As new **CPython** releases regularly introduce new `opcodes`, consuming a large number of the undefined `opcodes` for CMQ is undesirable. Thus, we cannot introduce a new `opcode` for each optimized derivative a C extension provides. Instead, we define one additional `opcode` for each *optimizable* generic instruction. In other words, we add one `opcode` for each generic instruction for which a C extension can provide one or many optimized derivatives. For example, we add the `BINARY_OP_EXTERNAL` `opcode` since C extensions can specialize `BINARY_OP`.

One additional `opcode` is not sufficient, however, to differentiate between different derivatives. For example, our modified **NumPy** adds several derivatives for `BINARY_OP`, depending on the operation and the operand types involved. To that end, when C extensions register their specialized derivatives, CMQ assigns each derivative for the same instruction a unique id. CMQ stores the ids in a table to map each id to an external `opcode` handler. During specialization, CMQ repurposes the `oparg` of the corresponding `*_EXTERNAL` instruction to hold the id and, thus, to identify the exact derivative. For example, assume that **NumPy** wants to specialize an occurrence of `BINARY_OP` with a derivative `NP_ADD_FLOAT_FLOAT`. During the initial registration, CMQ assigns the derivative `NP_ADD_FLOAT_FLOAT` the id 5. During specialization, CMQ replaces the generic `BINARY_OP` with `BINARY_OP_EXTERNAL` and its original `oparg` with 5. During execution of the `BINARY_OP_EXTERNAL` occurrence, CMQ looks up the external `opcode` handler with the `oparg` and calls the external handler.

This approach has advantages as well as disadvantages and is specific to **CPython**'s internal implementation. One advantage is that this approach consumes only a small number of `opcodes`. Another advantage is that CMQ has to rewrite only the replaced instruction, as opposed to multiple instructions affected by a layout change. Since the `*_EXTERNAL` instructions have the same inline cache size as their generic counterparts, the layout of the instructions remains the same. If CMQ instead, e.g., changed the inline cache size, all jumps crossing the affected instruction as well as exception-handling tables would have to be rewritten.

A disadvantage of this approach is that it introduces an additional indirection. The `opcode` handlers of the `*_EXTERNAL` instructions have to lookup the external function with the `oparg`. For **CPython** with **NumPy** we found this overhead to be negligible and prioritized the benefit of saving `opcode` space. In language VMs with a larger `opcode` space, or in cases where the indirection negatively affects performance, specialized derivatives can be mapped directly to `opcodes`. A hybrid approach is possible as well. For example, particularly performance-critical derivatives can receive their own `opcode`, whereas other derivatives are grouped according to the scheme above.

5.2.4 Implementing Extension-Delimited Superinstructions

In Section 4.2.2 we discussed the concept of extension-delimited superinstructions. We implemented extension-delimited superinstructions by allowing C extensions to indicate unused arguments during specialization. For example, our modified NumPy specializes `BINARY_SUBSCRIPT` by precomputing its index datastructure and replacing it with a `NP_BINARY_SUBSCRIPT_CONSTANT` derivative (see Section 6.4). As a result, all the index operands required by `BINARY_SUBSCRIPT` become unused. CMQ-NumPy marks the operands as unused via the `OINT` and CMQ automatically takes care of skipping the operand setup during later executions.

In a first step, CMQ determines the instructions responsible for pushing the unused operands onto the stack. We call these instructions `operand originators`. As the `operand originators` are no longer needed, their operands become unused as well. In a second step, CMQ recursively finds the `operand originators` of the now unused operands. This process continues until CMQ has found the first unused instruction in the sequence. CMQ then replaces the first instruction with a `JUMP` that jumps directly to the optimized derivative, e.g., `NP_BINARY_SUBSCRIPT_CONSTANT`. Note, however, that such an optimization is only possible if the skipped instructions are side-effect-free. If, for example, one of the instructions is a `CALL` instruction, CMQ does not optimize the argument setup.

5.2.5 Deoptimization in CPython

As optimization assumptions can become invalid, CMQ needs a way to restore the original instructions in such a case. To that end, CMQ records a `deopt structure` for each instruction optimized. The `deopt structure` contains a pointer to the optimized instruction and the original `opcode` and `oparg`. The approach described in Section 5.2.3 requires CMQ to replace the original `oparg` during specialization. The backup copy in the `deopt structure` enables CMQ to restore the `oparg` upon deoptimization. Once the original instruction is restored, CMQ executes the original instruction instead of the derivative.

For extension-delimited superinstructions (see Section 4.2.2), the `deopt structure` stores the entire list of instructions that were replaced with the superinstruction. When deoptimizing extension-delimited superinstructions it is not enough to restore the original instructions, however. Once the language VM reaches the deoptimizing extension-delimited superinstruction, the instruction pointer is already past the instructions that would have pushed the operands to the stack (see Section 5.2.4). Since the extension-delimited superinstruction does not expect the same number of stack operands as the original instruction, executing the original instruction would fail. Thus, after deoptimizing an extension-delimited superinstruction, CMQ replays all instructions responsible for the stack operands of the restored instruction. Replaying is possible because we limit the related optimization to side-effect-free instructions (see Section 5.2.4).

6 Implementation of Cross-Module Quicken in NumPy

To demonstrate the optimizations enabled by CMQ, we extended NumPy to use the OINT and implemented various optimized derivatives. On module initialization, CMQ-NumPy registers its optimization hook with CPython and later optimizes instructions related to array operations. To understand these optimizations we first give a short overview of NumPy in Section 6.1. In Sections 6.2–6.4 we outline how we implemented the CMQ-NumPy optimizations.

6.1 NumPy in a Nutshell

NumPy is one of the most popular CPython C extensions and consistently among the top 20 downloaded PyPi packages [22]. The NumPy package provides multidimensional arrays, called *ndarrays*, of different data types that optionally can be contiguous, aligned and iterated in different iteration orders. In addition to data representation via arrays, NumPy also contains a variety of mathematical functions operating on those arrays. NumPy is also a cornerstone of several other CPython packages, such as Pandas, SciPy, scikit-learn and PyTorch. To integrate seamlessly with Python, NumPy makes extensive use of operator overloading and, e.g., allows to add, subtract, multiply or divide arrays. Behind the scenes, NumPy takes care of transforming the arrays as necessary to perform the desired operation. For example, through a mechanism called *broadcasting*, NumPy allows to transparently add two arrays with a different number of dimensions:

```
>>> np.array([1, 2, 3]) + np.array([[5, 6, 7], [1, 2, 3]])
array([[6, 8, 10],
       [2, 4, 6]])
```

NumPy implements many of these operations on *ndarrays* as so called *universal functions* or *ufunc* for short. A *ufunc* object represents a mathematical function that operates element-wise on *ndarrays*. Each *ufunc* can have multiple underlying implementations of the mathematical function, called *array methods*. Which array method a *ufunc* uses depends, among other factors, on the input operand types. Internally, NumPy implements array methods as tuned C loops to exploit available hardware features (e.g., vectorization). Before calling any array method, *ufuncs* are responsible for type casting, broadcasting and several other standard NumPy features.

NumPy determines the *ufunc* and subsequently the array method responsible for performing an *ndarray* operation in a multistep process. First, NumPy determines the responsible *ufunc* object. For binary operations with operator overloading, NumPy reads the *ufunc* from a module-wide table. For other operations, such as `minimum` or `maximum`, the *ufunc* object is a callable Python object and pushed onto the operand stack. The subsequent steps are identical for both cases, and we summarize them in List 2.

6.2 Exploiting ufunc Type Stability

NumPy's flexibility and extensibility has allowed it to become a building block in a number of different domains. For example, NumPy allows users to customize almost any step in List 2. This flexibility comes at a cost, however. For every array addition, CPython first performs the steps in List 1 and then the steps in List 2. A crucial observation is that many of the steps in List 2 can be eliminated or simplified by fixating the types and number of inputs to the *ufunc*. For example, when adding exactly two arrays in a `BINARY_OP`, the following simplifications are possible.

If both input arrays are of type *ndarray*, Step 1 and Step 5 become redundant. If, in addition, the array element types are known, Step 3 becomes redundant. Type-specialized instructions described in Section 4.2.1 allow CMQ to efficiently speculate on these properties. By additionally speculating that the user has not changed the default *ufunc* for adding arrays, Step 2 and Step 3 become redundant. CMQ enables this type of speculation with the deoptimization strategies outlined in Section 4.1.1.

1. Check if any of the operands overrides the `ufunc`. NumPy allows any operand participating in a `ufunc` operation to override the responsible `ufunc` object, effectively implementing a form of multi-dispatch;
2. Determine the exact casting rules and perform any necessary casting. For example, in this step NumPy converts scalar values participating in an array operation into arrays;
3. Based on the resulting types from the previous step, resolve the array method;
4. With the array method, resolve the operation types, in particular the result type;
5. Call array preparation functions, if any;
6. Check if a single iteration of the array method loop is possible by analyzing the properties of the participating arrays. Such a simplified case is possible for certain configurations of input arrays. We skip the exact details here for brevity.
7. If a single loop is sufficient, allocate the output array (if necessary) and call the array method loop
8. Otherwise, allocate an iterator and call the array method as many times as dictated by the iterator.

■ List 2 Steps for resolving NumPy `ufunc` and array methods. For more details see the NumPy Enhancement Proposals 13 and 18 [8, 27], the NumPy manual on `ufuncs` [19] and the function `ufunc_generic_fastcall` in `ufunc_object.c` in the NumPy codebase.

To unlock these optimizations, CMQ-NumPy provides specialized `BINARY_OP` derivatives for several array type combinations. For example, CMQ-NumPy specializes `BINARY_OP` occurrences that add or subtract two float arrays, effectively eliminating Step 1–5. While the case distinction in Step 6 and the last step (either Step 7 or Step 8) remain, the specialized derivatives simplify Step 6 to a few comparisons. In the original NumPy, Step 6 is handled by a function that needs to handle several corner cases and deal with potentially more than two input arguments. The added assumptions in the derivatives allow us to partially evaluate the function and to inline the remaining checks directly into the derivatives. As the optimized derivative is represented as `BINARY_OP_EXTERNAL` in CPython (see Section 5.2.3), the optimization also eliminates the `BINARY_OP` dispatching steps (see List 1).

A similar optimization is possible for calls of `ufuncs` objects via `CALL` instructions. As an example, consider a call to the `minimum` function of the NumPy package: `numpy.minimum([1, 2], [3, 4])`. CPython first loads the `minimum` `ufunc` object from the NumPy module and pushes the object to the stack. Next, CPython pushes the argument lists onto the stack. Finally, CPython calls the `ufunc` object via the `Vectorcall` protocol for calling into C extensions. Like for the `BINARY_OP` instructions, CMQ-NumPy provides a derivative that skips many of the steps in List 2 and calls the appropriate array method directly. During specialization, CMQ-NumPy not only validates the operand types, but also ensures that the `ufunc` object represents the expected `minimum` function. Once specialized, the loading of the `ufunc` object becomes redundant (see Section 4.2.2).

6.3 Automatic Generation of Derivatives

During the implementation of `BINARY_OP` derivatives, we noticed that the code of different derivatives differs only at select locations. Specifically, each derivative validates its type-specific assumptions and calls a type-specific array method. All other aspects of the code, such as the simplified Step 6 are identical between the derivatives. For example, all derivatives analyze certain properties of the input arrays, such as dimensions and strides, to decide whether Step 7 or Step 8 is necessary. Similarly, the code to decide whether a derivative is suitable for an instruction occurrence differs only in details.

```

BinOp(
    operation="add",
    left_type="adouble",
    right_type="adouble",
    result_type="NPY_DOUBLE",
    loop_function="DOUBLE_add",
    commutative=True,
)

```

(a) Specification of a derivative that adds two double arrays.

```

if((PyArray_CheckExact(lhs) &&
PyArrayHasType(NPY_DOUBLE) &&
PyFloat_CheckExact(rhs)) ||
// symmetrical commutative case
{
// Specialize for adding
// double arrays
}

```

(b) Automatically generated condition for specializing float array addition. The highlighted parts are taken from the derivative description.

Figure 5 Derivative description (left) and the automatically generated specialization condition (right).

To reduce code duplication, we wrote a code generator in Python that uses Mako templates to generate the various cases and derivatives. The code generator takes a specification of the derivatives produces specialization conditions and derivative implementations. Figure 5a shows an example of the double-array addition derivative specification. The specification defines the required types and the concrete array method to use in the derivative implementation. The code generator automatically generates derivative implementations and their corresponding specialization conditions. Figure 5b shows an example of a generated condition. Since addition is a commutative operation, the code generator automatically generates the symmetric case as well.

The code generator not only reduced the amount of duplicate code, but also allowed us to experiment with different implementation variants. For example, we tested a variant that forcefully inlines all function calls within the derivative implementations and found the performance difference to be negligible.

6.4 Per-Instruction Caches in NumPy

In Section 4.2.3 we described how CMQ enables an instruction-occurrence-specific caching via an `occurrence cache`. CMQ-NumPy uses the `occurrence cache` in optimized `BINARY_OP` and `BINARY_SUBSCRIPT` derivatives. Specifically, the `occurrence cache` we implemented in the OINT in CPython allows CMQ-NumPy to store a pointer for each optimized instruction.

The `BINARY_OP` derivatives use the `occurrence cache` keep a scratch array for results. The idea is based on the observation that `BINARY_OP` instructions often allocate short-lived arrays for the operation result and, thus, cause pressure on the memory subsystem. We gauge the effectiveness of the `occurrence cache` in Section 7.4. Whenever our optimized `BINARY_OP` derivatives allocate a new result array, they store a pointer to the array in the `occurrence cache`. In subsequent executions, the derivatives try to reuse the cached array instead of allocating a new one. Reusing a cached array is possible whenever the cached array has a reference count of 1, meaning that the cache is the only reference to the object. The result cache trades memory for CPU cycles by avoiding the recurring allocation and deallocation of frequently used objects.

In contrast, the `BINARY_SUBSCRIPT` derivatives use the `occurrence cache` to store information precomputed at specialization time. In CPython, a `BINARY_SUBSCRIPT` instruction uses a subscript object to access subscriptable, such as lists or NumPy arrays. NumPy extends

CPython’s subscripting mechanism with the notion of multidimensional subscripts. For example, the expression `array[1:, 2:]` selects all sub-arrays beginning at the second and from each selected subarray all elements beginning at the third. Under the hood, the expression `[1:, 2:]` is a syntactic sugar for `[(slice(1, None, None), slice(2, None, None))]`. In other words, the subscript object is a tuple consisting of two slice objects. While this syntax is highly expressive and makes it easy to navigate nested arrays, the flexibility comes at a cost. For every such subscript access, CPython needs to construct the participating objects, i.e., the slices and the tuple, and then call NumPy to handle the subscript on a NumPy array. The subscript object construction alone constitutes 7 instructions. Next, NumPy needs to deconstruct the subscript object again to compute an index structure that is later used to access the array. Similar to the case of resolving ufuncs (see List 2), the computation in NumPy is generic and needs to handle several corner cases.

An important observation is that all objects participating in the above subscript operation, except the array, are constant. To that end, CMQ-NumPy move the computation of the index structure from instruction execution to specialization time. During specialization of a `BINARY_SUBSCRIPT` instruction, CMQ-NumPy analyzes the instructions constructing the subscript object. If the subscript object is constant, CMQ-NumPy precomputes the index structure and stores a pointer to the structure in the occurrence cache. Instead of recomputing the structure, the specialized `BINARY_SUBSCRIPT` derivatives read the index structure from the cache.

7 Evaluation

7.1 System Configuration

Our changes are based on CPython 3.12.0 and NumPy 1.26.4. To guarantee a fair comparison and equal compilation parameters, we also built the baseline, i.e., CPython 3.12.0 and NumPy 1.26.4, from source.

We perform our evaluation on three different machines, summarized in Table 2. Machine EPYC is equipped with an AMD EPYC Rome 7H12 CPU running at 3.2 GHz, 1TB DDR4 RAM running at 3200 MHz, and Debian 12. Machine i7 is equipped with an Intel Core i7-8559U CPU running at 2.7 GHz, 64GB DDR4 RAM running at 2667 MHz, and Debian 12. Machine M3 is equipped with an Apple 16 core M3 CPU running at 4.05 GHz, 128GB RAM, and macOS 14. On each machine we compiled CPython and NumPy with the bundled GCC (12.2.0) and GNU linker (2.40).

7.2 Experimental Design

CMQ consists of a modified CPython instance that supports the Optimization Interface and a modified NumPy package that leverages the Optimization Interface.

We evaluate the performance improvements of CMQ based on the NPBench benchmark framework [41]. NPBench includes compute-intensive NumPy benchmarks and aims to compare the performance of NumPy-specific optimizing compilers. While our technique is not NumPy-specific, these benchmarks allow us to properly evaluate the afforded performance improvement. In addition to the benchmarks already included with NPBench, we integrated NumPy Phoronix benchmarks from openbenchmarking [33]. Like the included benchmarks, the Phoronix benchmarks consist of scientific kernels that make intensive use of NumPy.

NPBench supports differently sized input presets for the included benchmarks. For our evaluation, we used the `paper` preset, which was also used during the evaluation of NPBench itself [41]. The Phoronix benchmarks have their input sizes hardcoded into the benchmark.

We run all benchmarks with the `NPBench` test runner. The runner starts each benchmark in a new CPython process and repeats the benchmark a given number of times with the CPython `timeit` package. In addition, `NPBench` verifies that the results of an optimized implementation and the NumPy default implementation are equal. We modified `NPBench` to use our customized CPython and NumPy while measuring CMQ’s performance.

To reduce noise, we limit NumPy to a single thread and pin the benchmark run to a single CPU with `cset`. Note that this restriction does not influence CMQ’s relative performance improvement over standard NumPy. Distributing workloads to multiple threads happens in NumPy components unaffected by CMQ. We verified this experimentally by comparing runs with and without threading and found the differences to be within measurement noise (2-3%). Without limiting the number of threads (e.g., to 16) in our experiments, NumPy used *all* available logical CPUs, even for trivial tasks. For the EPYC Rome machine this meant distributing tasks to 256 logical CPUs, effectively overloading the machine synchronization overhead.

We repeat each `NPBench` benchmark 20 times and limit the execution time of a single run to 120s. Since the Phoronix are short-running, we repeat each benchmark run 100 times. We kept the internal iteration count of 40 for the Phoronix benchmarks. With this configuration, one benchmark (`3mm`) timed out in the baseline on all machines.

7.3 Performance

Figure 6 and Figure 7 shows the performance improvement of CMQ over the baseline for the `NPBench` and `Phoronix` benchmarks, respectively. Due to space constraints, we show only benchmarks where CMQ-NumPy could specialize at least one instruction. We give a complete list of benchmark results in Appendix A.

Whereas some `NPBench` benchmarks, such as `adist`, show no improvement, CMQ improves the performance of other benchmarks by a factor of up to 2.84. The improvements are similar on different machines, with the notable differences of `heat3d` and `floydwar`. On these two benchmarks, CMQ achieves no measurable performance improvement on M3. For the `NPBench` benchmarks, we report a geometric mean improvement for the machines EPYC, i7, and M3 of 1.11x, 1.10x and 1.08x, respectively.

For the `Phoronix` benchmarks the situation is similar. Some benchmarks, such as `periodic_dist`, show an improvement of up to 1.94, whereas other benchmarks, such as `eucl_dist` show no improvement. One difference to the `NPBench` benchmarks is that certain benchmarks show a slight decrease in performance, most notably `pairwise` and `rosen`. For the `Phoronix` benchmarks, we report a geometric mean improvement for the machines EPYC, i7, and M3 of 1.10x, 1.08x and 1.06x, respectively.

We discuss these differences in Section 8.1.

7.4 Dynamic Locality Analysis

To analyze type locality and cache stability, we collected various statistics on the EPYC machine over all `NPBench` benchmarks with 20 repetitions. We found the operand types on which the specialized derivatives speculate to be 100% stable except for `resnet`. In `resnet`, 3 operations had to deoptimize due to a changed operand type.

Table 1 shows relevant metrics for the `BINARY_SUBSCRIPT` result cache (see Section 6.4). The other derivatives (e.g., `BINARY_SUBSCRIPT`) cache only static data (e.g., the computed index structure) and, therefore, never need to invalidate the cache. For brevity, Table 1 shows only benchmarks in which cache invalidations occurred. `REFCNT` means the cached array had a reference count greater than one. `SHAPE` means the array did not have the expected shape.

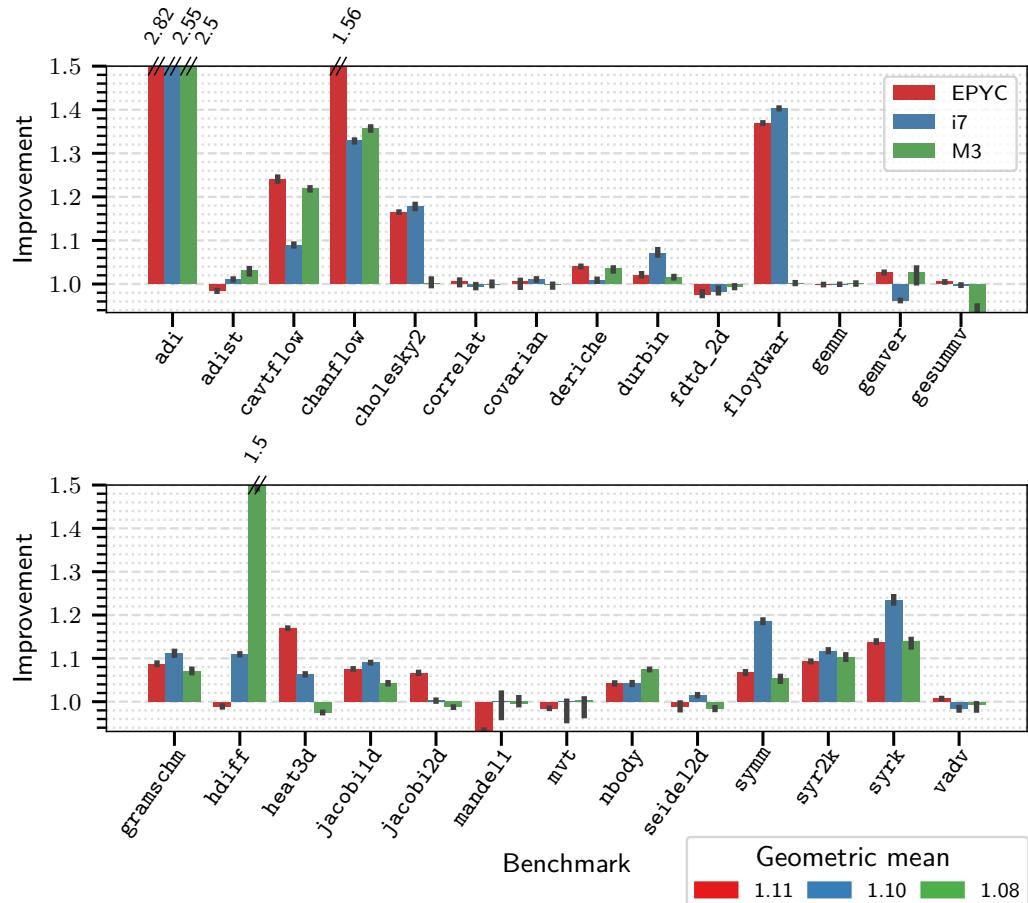


Figure 6 The performance improvement of CMQ-NumPy over the baseline for NPBench benchmarks with at least one specialized instruction. The black lines at the top of the colored bars show the 95% bootstrapping confidence interval with 1000 samples. For the bars that do not fit within the figure, a label on top of the bar shows their value.

In `cavtflow`, `chanflow` and `heat3d` a cached array had a reference count greater than 1, indicating that the array is not in fact temporary. In `syr2k`, the cached array's properties did not match the properties required for the result. CMQ-NumPy keeps a cache counter to detect cases where the cache is invalidated frequently and disables an instruction cache after 100 invalidations. The counter disabled the cache in `syr2k` for one instruction and in `vadv` in 8 instructions. We found this optimization to improve `cavtflow`'s performance by about 10%.

7.5 Implementation Effort

The changes in CPython consist of 1,136 insertions and 51 deletions across 27 files. These changes include the code for statistics, debugging routines, comments and newlines, but exclude files generated by the CPython build. The OINT consists of a hook used by C extensions to register, two callbacks provided by the C extension and three functions the C extension can use to specialize instructions.

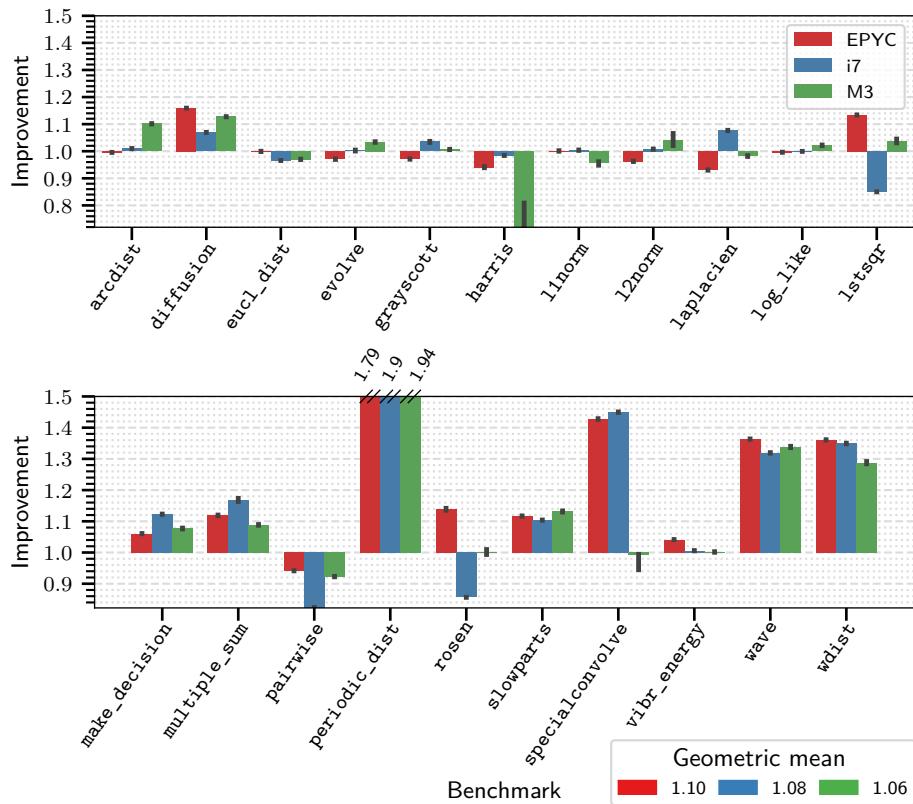


Figure 7 The performance improvement of CMQ-NumPy over the baseline for Phoronix benchmarks with at least one specialized instruction. The black lines at the top of the colored bars show the 95% bootstrapping confidence interval with 1000 samples.

The changes in NumPy consist of roughly 1200 lines of C, 400 lines of Python and 900 lines of template code. These changes include the code for statistics and performance measurements, derivative templates, debugging routines, comments and newlines, but exclude generated files. Implementing these changes took us roughly 3 months, with no prior experience with NumPy. NumPy consists of roughly 163,000 lines of code, which means that our extension (the C and template code) comprise less than 1% of NumPy’s code base. The template code for our optimized derivatives contains primarily rearrangements of existing NumPy code (e.g., applying a `ufunc` to an array with an iterator).

8 Discussion

This section discusses the evaluation results, in particular the varying performance results, as well as what we believe to be relevant threats to validity.

8.1 Performance

Figure 6 and Figure 7 detail the performance results obtained on three different CPU architectures. Although the performance is promising in some cases, it is indistinguishable from measurement noise in other cases. A closer look to what happens under the hood is required to analyze these differences.

Table 1 CMQ-NumPy result cache statistics (see Section 7.4).

Benchmark	Misses	Reason
cavtflow	308	REFCNT
chanflow	308	REFCNT
heat3d	132	REFCNT
syr2k	100	SHAPE
vadv	844	REFCNT

Table 2 Configuration of the benchmarking machines used in Section 7.3.

Machine	CPU	RAM
EPYC	AMD EPYC 7H12	1 TB
i7	Intel Core i7-8559U	64 GB
M3	Apple M3 Max	128 GB

An analysis of the executed interpreter instruction frequencies shows that CMQ-indifferent benchmarks execute *fewer* interpreter instructions. This difference also reduces the impact of CMQ optimizations. The `adi` benchmark, for example, executes most of its instructions in the `kernel` function, with each interpreter instruction executed about 20,000 times, with 20 iterations. In the `fdtd_2d` benchmark, on the other hand, the comparative interpreter execution count is only 500 times. This order-of-magnitude difference provides part of the answer.

The interpreter instruction execution frequency aside, the `fdtd_2d` benchmark provides another part of the answer. With the `paper` preset, `fdtd_2d` operates on large matrices having 1,000 rows of 1,200 columns. With an element size of a double floating point number, such a matrix spans 9,600,000 bytes, which is roughly 9 megabytes. Since this size exceeds the limits of both most operating system page sizes, and CPU data caches, the overall execution time is dominated by these caching effects.

To demonstrate the effect of these two variables on CMQ’s optimization potential, we manually changed the parameters of `fdtd_2d`. Instead of 500 repetitions on 1,000 by 1,200 matrices, we experimented with 10,000 iterations on 200 by 220 matrices. With these parameters, performance improved by about 20%.

On the Phoronix benchmark set (Figure 7), CMQ’s impact is less than on the NPBench benchmarks (Figure 6). The primary reason is that many of the Phoronix benchmarks operate on types for which we have not yet added optimized derivatives, such as 32bit floats and NumPy’s scalar types. In other words, these benchmarks pay the small, but non-zero price of attempted specializations without profiting from CMQ. The futile specialization attempts are also the reason for a slight *decrease* in performance for e.g., the `pairwise` benchmark. Compared to the NumPy benchmarks, the Phoronix benchmarks are short-running. As a result, the overhead of specialization attempts is high compared to the benchmark runtime. We confirmed this theory by increasing the internal iteration count, such that a single benchmark run takes longer. We found that with longer run times, the slowdown for all but one benchmarks approached zero. Only the slowdown of `grouping` remained at roughly 10%. The slowdown remained even when disabling specialization attempts entirely and the exact cause requires further analysis.

8.2 Implementation Effort

8.2.1 CPython

Integrating an CMQ into a language VM consists of two tasks. First, allowing C extensions to register and subsequently calling the C extension to attempt the specialization of hot instructions (see Section 5.2). Second, providing functionality to the C extension via the OINT to analyze, specialize and deoptimize instructions.

In the case of `CPython`, we could reuse much of `C`Python’s optimization-counter infrastructure to trigger the optimization of hot instructions Section 5.2.1. As a result, the first task, amounted to only about 200 lines of code. The OINT functionality for the second task consists of analyses (e.g., for finding the originator of an argument, see Section 5.2.4) and code for handling the optimization and deoptimization. The implementation of the OINT made up the majority of the implementation effort in `C`Python and amounts to roughly 800 lines of code.

8.2.2 NumPy

In general, the implementation effort for implementing optimizations depends largely on the C extension in question. As non-experts in NumPy, we spent the majority of the implementation time (see Section 7.5) with understanding NumPy’s architecture as well as debugging our implementation errors. We believe that domain experts (e.g., NumPy core developers) could implement the optimizations not only in substantially less time, but also with less code. For our research prototype we explicitly specified each derivative (see Section 6.3), leading to a larger amount of boilerplate code. Instead, developers with an intimate understanding of NumPy could generate the specifications from the `ufunc` operation specifications already present in NumPy. Future research could focus on automating parts of the optimization implementation, and thus reducing burden on C extension authors.

8.3 Threats to Validity

Although we spent a great deal of effort on making sure that both design/implementation and evaluation are unbiased and representative of the general principle explored and demonstrated by CMQ, the following threats to validity apply.

8.3.1 Generalization Beyond Python

Our analysis and findings focus on the `C`Python ecosystem. Although we believe that these findings hold equally well for similar ecosystems, such as Lua, Ruby, or even WASM, only a comparative investigation will be able to close this gap. Note that neither our analysis, nor our implementation, rely on specifics of the Python interpreter. Python, for example, uses a stack-based virtual machine interpreter architecture. Our extension-delimited superinstructions observation and optimization (cf. Section 4.2.2) hold equally well for register-based architectures.

The standard³ Ruby interpreter YARV is architecturally similar to Python. Both are written in C, both have bytecode interpreters, and although the YARV does not currently perform runtime specialization, a prototype for a specializing interpreter exists [30]. We thus believe that porting CMQ to Ruby would be relatively straightforward.

Another language VM with C extension support is the Lua VM and its optimized variant LuaJIT. The LuaJIT VM has both a profiling interpreter and a JIT compiler and retains compatibility with Lua C extensions. Unlike Python and Ruby, however, the LuaJIT VM is register-based and the interpreter is written in assembly. While certainly possible, the different architecture and low-level nature of the LuaJIT interpreter would pose an obstacle to porting CMQ to Lua.

³ As with Python, many different Ruby implementations exist. With “standard” we are referring to the interpreter that is part of the official Ruby distribution.

8.3.2 Generalization Beyond NumPy

Based on the domain-specificity of C extensions (cf. Section 3.1), our findings cannot translate to other C extensions *verbatim*. The qualitative analysis results apply in general (cf. Section 3.2), and also to other C extension ecosystems. The corresponding optimization techniques explored and demonstrated for the extenders-category also translate to other C extensions.

Our analysis for `lxml` Python extension indicates, for example, that `lxml` would benefit from extension-delimited superinstructions that operate on native types. Note in this context that our OINT design and implementation is not *closed*, but can be extended for other use cases, and indeed we expect future work, also by other researchers, to uncover more optimization features.

8.3.3 Performance Bias Through NPBench

We evaluate CMQ with NPBench that consists of a suite of compute-intensive scientific kernels. These benchmarks cannot be representative of other workloads for different C extensions. No claim to the expected speedup potential can be made on a sound scientific basis.

8.3.4 Performance Result Interpretation

The authors are not experts in optimization of mathematical kernels. The reported results are, thus, merely indicative. An expert possessing the relevant domain expertise may see, and actually uncover, more optimization potential.

9 Related Work

In the Python ecosystem, `Numba` is one way to speed up scientific Python programs, in particular programs using `NumPy`. `Numba` is a Python JIT compiler based on the LLVM JIT compiler framework [28]. As shown by Ziogas et al., `Numba`'s JIT-approach enables impressive performance improvements for some benchmarks [41]. However, `Numba` supports only a subset of Python and cannot optimize functions with incomplete type information. `Cython` is a compiler that compiles a superset of Python to optimized C code and aims to narrow the gap between writing Python code and C extensions [3]. In addition to lowering the burden of writing C extensions, an extension to `Cython` could help to automatically generate optimized derivatives for CMQ.

Grimmer et al. take a different approach to dealing with C extensions [25]. Their Truffle Multi-Language Runtime runs both, the host language and the C extension, on the same language VM, on top of the Truffle framework. Running the C extension is possible through a C interpreter implemented in Truffle [23]. In lack of a benchmark suite for C extensions, the authors evaluate the peak performance of the Multi-Language Runtime with two Ruby programs. A later paper suggests that the performance depends on the exact language combination and benchmark [24]. The approach of running C extensions with a Truffle C-Interpreter was later generalized with `Sulong` [32].

The work closest to ours is “Dr Wenowdis”, a system to communicate function type information from C extensions to PyPy [7]. In their paper, the authors focus primarily on boxing and unboxing overhead, but the principles are similar to our type-specialized instructions (see Section 4.2.1). We believe that our work is mutually beneficial with “Dr Wenowdis” and that the principles of CMQ could be extended to JIT compilers as well.

The WebAssembly Garbage Collector (WASM GC) proposal is similar in spirit to CMQ [26]. With WASM GC, a language implementation running on a WASM engine can communicate information about its object layout to the WASM host engine. This additional communication enables the WASM garbage collector to reason about and to collect guest objects. Thus, the guest language implementation is no longer a black box to the WASM host engine. While WASM does not have C extensions, the proposed WASM System Interface (WASI) fulfills a similar purpose. We believe, therefore, that CMQ’s principles could benefit WASI as well.

10 Conclusions

We present the first analysis and exploration of C extensions for dynamic languages, exemplified by the Python ecosystem. Based on this analysis, we find that the key obstacle of a large-scale quantitative analysis is that many C extensions require their own *domain expertise*. This domain specificity of C extensions makes them both difficult to compare and difficult to evaluate performance against, since the domain specificity also implies a lack of generalizable benchmark suites.

Due to this negative result, we instead focus on a qualitative analysis of Python’s C extension ecosystem. We find that C extensions fall into three categories: (i) optimizers, (ii) binders, and (iii) extenders. Optimizers are C extensions that could be written in Python, but are written in C to speed up the processing. Binders are C extensions that essentially bind Python to existing C libraries. Extenders add functionality to Python that does not readily exist.

From a performance perspective, we find that the first two categories provide few optimization opportunities. This lack of opportunities is rooted in the fact that most time is spent in the C extensions themselves. The third category, however, offers optimization potential as evidenced by the speedups demonstrated by CMQ. Based on the example of NumPy, we illustrate a total of three orthogonal optimization techniques.

Since our work represents, to the best of our knowledge, the first foray into optimization across module boundaries, we expect future work efforts that extend and generalize the ideas presented herein. We believe that a natural step would be to try integrating our findings into just-in-time compilers. A generalization, on the other hand, would try to apply our ideas to another dynamic language ecosystem, such as Ruby, Lua, PHP, or Perl. We furthermore expect that the presented system will be adapted and extended by performance-conscious extension authors, leading to new optimization opportunities down the road. Finally, even a closed ecosystem such as JavaScript may benefit from our ideas: the runtime system and the browser represent a form of C extension for the JavaScript virtual machine. Through similar APIs, JavaScript engines could, thus, benefit from optimizations.

References

- 1 Scott B. Baden. High Performance Storage Reclamation in an Object-Based Memory System. Technical Report, University of California at Berkeley, USA, May 1982.
- 2 Gergö Barany. Python interpreter performance deconstructed. In *Proceedings of the Workshop on Dynamic Languages and Applications, Dyla 2014, Edinburgh, United Kingdom, June 9-11, 2014*, pages 5:1–5:9, Edinburgh United Kingdom, June 2014. ACM. doi:10.1145/2617548.2617552.
- 3 Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcín, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Comput. Sci. Eng.*, 13(2):31–39, March 2011. doi:10.1109/MCSE.2010.118.

- 4 Felix Berlakovich. CMQ CPython implementation. Software (visited on 2024-08-29). URL: <https://github.com/fberlakovich/cmq-ae>.
- 5 Felix Berlakovich. CMQ Numpy implementation. Software (visited on 2024-08-29). URL: <https://github.com/fberlakovich/cmq-numpy-ae>.
- 6 Felix Berlakovich and Stefan Brunthaler. Cross-Module Quickening. Software (visited on 2024-08-29). URL: <https://doi.org/10.5281/zenodo.11174717>.
- 7 Maxwell Bernstein and CF Bolz-Tereick. Dr wenowdis: Specializing dynamic language C extensions using type information. *CoRR*, abs/2403.02420(arXiv:2403.02420), March 2024. doi:10.48550/arXiv.2403.02420.
- 8 Blake Griffith. A mechanism for overriding Ufuncs. URL: <https://numpy.org/neps/nep-0013-ufunc-overrides.html>.
- 9 Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. Allocation removal by partial evaluation in a tracing JIT. In Siau-Cheng Khoo and Jeremy G. Siek, editors, *Proceedings of the 2011 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2011, Austin, TX, USA, January 24-25, 2011, PEPM ’11*, pages 43–52, New York, NY, USA, January 2011. ACM. doi:10.1145/1929501.1929508.
- 10 Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. Runtime feedback in a meta-tracing JIT for efficient dynamic languages. In Ian Rogers, Eric Jul, and Olivier Zendra, editors, *Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems, ICOOOLPS 2011, Lancaster, United Kingdom, July 26, 2011, ICOOOLPS ’11*, pages 9:1–9:8, New York, NY, USA, July 2011. ACM. doi:10.1145/2069172.2069181.
- 11 Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: Pypy’s tracing JIT compiler. In Ian Rogers, editor, *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, ICOOOLPS 2009, Genova, Italy, July 6, 2009, ICOOOLPS ’09*, pages 18–25, New York, NY, USA, July 2009. ACM. doi:10.1145/1565824.1565827.
- 12 Stefan Brunthaler. Virtual-machine abstraction and optimization techniques. *Electronic Notes in Theoretical Computer Science*, 253(5):3–14, December 2009. doi:10.1016/j.entcs.2009.11.011.
- 13 Stefan Brunthaler. Inline caching meets quickening. 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 429–451, Berlin, Heidelberg, 2010. Springer. doi:10.1007/978-3-642-14107-2_21.
- 14 Stefan Brunthaler. Multi-level quickening: Ten years later. *CoRR*, abs/2109.02958, 2021. doi:10.48550/arXiv.2109.02958.
- 15 Lin Cheng, Berkin Ilbeyi, Carl Friedrich Bolz-Tereick, and Christopher Batten. Type freezing: exploiting attribute type monomorphism in tracing JIT compilers. In *CGO ’20: 18th ACM/IEEE International Symposium on Code Generation and Optimization, San Diego, CA, USA, February, 2020*, CGO 2020, pages 16–29, New York, NY, USA, February 2020. ACM. doi:10.1145/3368826.3377907.
- 16 Maxime Chevalier-Boisvert, Noah Gibbs, Jean Boussier, Si Xing (Alan) Wu, Aaron Patterson, Kevin Newton, and John Hawthorn. YJIT: a basic block versioning JIT compiler for cruby. In Gregor Richards and Manuel Rigger, editors, *VMIL 2021: Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages, Virtual Event / Chicago, IL, USA, 19 October 2021*, pages 25–32, Chicago IL USA, October 2021. ACM. doi:10.1145/3486606.3486781.
- 17 Maxime Chevalier-Boisvert, Takashi Kokubun, Noah Gibbs, Si Xing (Alan) Wu, Aaron Patterson, and Jemma Issroff. Evaluating yjit’s performance in a production context: A pragmatic approach. In Rodrigo Bruno and Eliot Moss, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, MPLR 2023, Cascais, Portugal, 22 October 2023*, MPLR 2023, pages 20–33, New York, NY, USA, October 2023. ACM. doi:10.1145/3617651.3622982.

- 18 L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In Ken Kennedy, Mary S. Van Deusen, and Larry Landweber, editors, *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, Salt Lake City, Utah, USA, January 1984*, pages 297–302, New York, New York, USA, 1984. ACM Press. ISSN: 07308566. doi:10.1145/800017.800542.
- 19 NumPy Developers. Universal functions (ufunc) basics – NumPy v1.26 Manual. URL: <https://numpy.org/doc/1.26/user/basics.ufuncs.html#type-casting-rules>.
- 20 M. Anton Ertl and David Gregg. The behavior of efficient virtual machine interpreters on modern architectures. In Rizos Sakellariou, John A. Keane, John R. Gurd, and Len Freeman, editors, *Euro-Par 2001: Parallel Processing, 7th International Euro-Par Conference Manchester, UK August 28-31, 2001, Proceedings*, volume 2150 of *Lecture Notes in Computer Science*, pages 403–412, Berlin, Heidelberg, 2001. Springer. doi:10.1007/3-540-44681-8_59.
- 21 M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In Ron Cytron and Rajiv Gupta, editors, *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, PLDI '03, pages 278–288, New York, NY, USA, May 2003. ACM. doi:10.1145/781131.781162.
- 22 Christopher Flynn. PyPI Download Stats. URL: <https://pypistats.org/top>.
- 23 Matthias Grimmer, Manuel Rigger, Roland Schatz, Lukas Stadler, and Hanspeter Mössenböck. Trufflec: dynamic execution of C on a java virtual machine. In Joanna Kolodziej and Bruce R. Childers, editors, *2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14, Cracow, Poland, September 23-26, 2014*, PPPJ '14, pages 17–26, New York, NY, USA, September 2014. ACM. doi:10.1145/2647508.2647528.
- 24 Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, and Mikel Luján. Cross-language interoperability in a multi-language runtime. *ACM Trans. Program. Lang. Syst.*, 40(2):8:1–8:43, May 2018. doi:10.1145/3201898.
- 25 Matthias Grimmer, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. Dynamically composing languages in a modular way: supporting C extensions for dynamic languages. In Robert B. France, Sudipto Ghosh, and Gary T. Leavens, editors, *Proceedings of the 14th International Conference on Modularity, MODULARITY 2015, Fort Collins, CO, USA, March 16–19, 2015*, pages 1–13, Fort Collins CO USA, March 2015. ACM. doi:10.1145/2724525.2728790.
- 26 WebAssembly Community Group and Andreas (editor) Rossberg. WebAssembly Core Specification. Technical report, W3C, 2024.
- 27 Stefan Hoyer, Matthew Rocklin, Marten van Kerkwijk, and Hameer Abbasi. A dispatch mechanism for numpy’s high level array functions. URL: <https://numpy.org/neps/nep-0018-array-function-protocol.html>.
- 28 Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A LLVM-based python JIT compiler. In Hal Finkel, editor, *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM 2015, Austin, Texas, USA, November 15, 2015*, LLVM ’15, pages 7:1–7:6, New York, NY, USA, November 2015. ACM. doi:10.1145/2833157.2833162.
- 29 Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, Mass., 1. print edition, 1997.
- 30 Vladimir Makarov. A Faster CRuby interpreter with dynamically specialized IR. URL: <https://rubykaigi.org/2022>.
- 31 Nagy Mostafa, Chandra Krintz, Calin Cascaval, David Edelsohn, Priya Nagpurkar, and Peng Wu. Understanding the Potential of Interpreter-based Optimizations for Python. Technical report, University of California, Santa Barbara, September 2010.
- 32 Manuel Rigger, Matthias Grimmer, and Hanspeter Mössenböck. Sulong – Execution of LLVM-based languages on the JVM: position paper. In *Proceedings of the 11th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems, ICOOLPS@ECOOP 2016, Rome, Italy, July 17-22, 2016*, ICOOLPS ’16, pages 7:1–7:4, New York, NY, USA, July 2016. ACM. doi:10.1145/3012408.3012416.

- 33 Victor Rodriguez Bahena. Numpy Benchmark Benchmark – OpenBenchmarking.org. URL: <https://openbenchmarking.org/test/pts/numpy>.

34 Christopher Graham Seaton. *Specialising dynamic techniques for implementing the Ruby programming language*. PhD thesis, University of Manchester, UK, 2015. URL: <https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.674722>.

35 Mark Shannon. *The construction of high-performance virtual machines for dynamic languages*. PhD thesis, University of Glasgow, UK, 2011. URL: <http://theses.gla.ac.uk/2975/>.

36 Tiobe. index ert TIOBE – The Software Quality Company, 2021. URL: <https://www.tiobe.com/tiobe-index/>.

37 Christian Wimmer and Stefan Brunthaler. Zippy on truffle: a fast and simple implementation of python. In Antony L. Hosking and Patrick Th. Eugster, editors, *SPLASH’13 - The Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity, Indianapolis, IN, USA, October 26-31, 2013*, pages 17–18, Indianapolis Indiana USA, October 2013. ACM. doi:10.1145/2508075.2514572.

38 Qiang Zhang, Lei Xu, and Baowen Xu. Regcpython: A register-based python interpreter for better performance. *ACM Trans. Archit. Code Optim.*, 20(1):14:1–14:25, March 2023. doi:10.1145/3568973.

39 Qiang Zhang, Lei Xu, Xiangyu Zhang, and Baowen Xu. Quantifying the interpretation overhead of python. *Sci. Comput. Program.*, 215:102759, March 2022. doi:10.1016/j.scico.2021.102759.

40 Wei Zhang, Per Larsen, Stefan Brunthaler, and Michael Franz. Accelerating iterators in optimizing AST interpreters. *ACM SIGPLAN Notices*, 49(10):727–743, December 2014. doi:10.1145/2660193.2660223.

41 Alexandros Nikolaos Ziogas, Tal Ben-Nun, Timo Schneider, and Torsten Hoefer. Npbench: a benchmarking suite for high-performance numpy. In Huiyang Zhou, Jose Moreira, Frank Mueller, and Yoav Etsion, editors, *ICS ’21: 2021 International Conference on Supercomputing, Virtual Event, USA, June 14-17, 2021*, ICS ’21, pages 63–74, New York, NY, USA, June 2021. ACM. doi:10.1145/3447818.3460360.

A All Benchmarks

■ **Table 3** All NPBench benchmark results.

■ **Table 4** All Phoronix benchmark results.

benchmark	EPYC	i7	M3	benchmark	EPYC	i7	M3	benchmark	EPYC	i7	M3
arc_dist.	1.00	1.01	1.10	l1norm	1.00	1.00	0.96	repeating	0.93	1.00	0.97
check_mask	0.98	1.07	0.98	l2norm	0.96	1.01	1.04	rev._cumsum	1.00	1.00	1.54
create_grid	1.05	1.00	1.00	laplacien	0.93	1.08	0.98	rosen	1.14	0.86	1.00
cronbach	0.97	0.96	0.97	local_max	0.97	0.96	0.98	slowparts	1.12	1.10	1.13
diffusion	1.16	1.07	1.13	log_like	1.00	1.00	1.02	spec.conv.	1.43	1.45	0.99
eucl-dist	1.00	0.97	0.97	lstsq	1.13	0.85	1.04	vibr_energy	1.04	1.01	1.00
evolve	0.97	1.00	1.03	make_dec	1.06	1.12	1.08	wave	1.36	1.32	1.34
grayscott	0.97	1.04	1.01	mult_sum	1.12	1.17	1.09	wdist	1.36	1.35	1.29
grouping	0.91	0.99	0.99	norm-comp	0.99	0.99	0.99				
harris	0.94	0.98	0.72	pairwise	0.94	0.82	0.92				
hasting	1.00	1.00	0.95	perio_dist	1.79	1.90	1.94				
Geomean	1.06	1.05	1.05								

HOBBIT: Hashed OBject Based InTegrity

Matthias Bernad 

μ CSRL – Munich Computer Systems Research Lab, Research Institute CODE,
University of the Bundeswehr Munich, Neubiberg, Germany

Stefan Brunthaler 

μ CSRL – Munich Computer Systems Research Lab, Research Institute CODE,
University of the Bundeswehr Munich, Neubiberg, Germany

Abstract

C vulnerabilities usually hold verbatim for C++ programs. The *counterfeit-object-oriented programming* attack demonstrated that this relation is asymmetric, i.e., it only applies to C++. The problem pinpointed by this COOP attack is that C++ does not validate the integrity of its objects. By injecting malicious objects with manipulated virtual function table pointers, attackers can hijack control-flow of programs. The software security community addressed the COOP-problem in the years following its discovery, but together with the emergence of transient-execution attacks, such as Spectre, researchers also shifted their attention.

We present HOBBIT, a software-only solution to prevent COOP attacks by validating object integrity for virtual function pointer tables. HOBBIT does not require any hardware specific features, scales to multi-million lines of C++ source code, and our LLVM-based implementation offers a configurable performance impact between 121.63% and 2.80% on compute-intensive SPEC CPU C++ benchmarks. HOBBIT’s security analysis indicates strong resistance to brute forcing attacks and demonstrates additional benefits of using execute-only memory.

2012 ACM Subject Classification Security and privacy → Software security engineering; Software and its engineering → Compilers

Keywords and phrases software security, code-reuse attacks, language-based security, counterfeit-object-oriented programming, object integrity, compiler security

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.7

Supplementary Material Software: <https://doi.org/10.5281/zenodo.11046716> [8]

Software: <https://github.com/mbernad/hobbit-artifact> [7]

Funding The research reported in this paper has been funded by the Federal Ministry for Climate Action, Environment, Energy, Mobility, Innovation and Technology (BMK), the Federal Ministry for Labour and Economy (BMAW), and the State of Upper Austria in the frame of the COMET Module Dependable Production Environments with Software Security (DEPS) [(FFG grant no. 888338)] within the COMET - Competence Centers for Excellent Technologies Programme managed by Austrian Research Promotion Agency FFG.

1 Motivation

Among the myriad of security exploits, control-flow hijacking is the most severe problem, as it allows the attacker to execute arbitrary code. A buffer overflow, for example, allows an attacker to overwrite the return address stored in a function’s stack frame, and thus divert control-flow to a location of her choice. Many other similar vulnerabilities exist and have been both explored and exploited over the past two decades. Most of these vulnerabilities affect both C and C++ alike.

The feasibility of an attack focusing exclusively on the C++ superset was demonstrated by Schuster et al. in 2015 [45]. By injecting malicious objects into a C++ application the attack hijacks control-flow and allows Turing-complete, arbitrary computation. In analogy to other similar attacks, such as return-oriented programming, this attack is known as *counterfeit-object-oriented programming*, COOP for short.

Due to the prevalence of C++ in systems and application software, researchers focused on devising mitigations against COOP. To prevent control-flow hijacking, prior defenses apply principles from effective C defenses. The principle of W^X limits attacker capabilities to inject code through new hardware features, such as Intel’s NX bits [52]. CFIxx, for example, uses the MPX extension to secure a bookkeeping table it relies upon [12]. By applying cryptography to encode and decode control-flow data, adversaries cannot know *a priori* how target addresses are encoded. CCFI, for example, uses Intel’s AES-NI instructions to cryptographically secure program addresses, such as return addresses, function-, and `vtable` pointers [31].

Although both of these defenses thwart COOP attacks, they, too, have drawbacks. Reliance on Intel’s MPX is problematic for three reasons. First, MPX may not be available in a system’s target environment, such as in embedded systems or IoT contexts. Second, Intel could decide to abandon the MPX instruction set extensions. Consider the MPK instruction set extension, which was discontinued rather abruptly, rendering defenses relying on it incapacitated. Third, MPX is non-compositional: A defense cannot protect an application that already relies on MPX for its business logic, as the MPX registers are already taken.

Cryptographic protection of pointers is desirable due to strong security guarantees, but suffers from prohibitive performance penalties. CCFI’s use of AES-NI reserves x86-64’s vector registers, i.e., SSE, AVX, AVX2, or AVX512, blocking their use for other purposes. Unavailability affects video processing, cryptographic operations, and a variety of other tasks.

HOBBIT neither requires specific hardware extensions nor blocks vector registers and, thus, addresses both of these challenges. Instead, HOBBIT modifies the C++ object layout to embed an integrity signature when an object is constructed. This signature is validated *before* executing each virtual method’s body.

A Clang/LLVM-based implementation of HOBBIT compiles large programs, such as the WebKit browser, and allows parameterization to balance security with performance. The key factor affecting performance is the choice of hashing technique to create an object’s signature. Our evaluation shows that choosing strong hashing techniques can lead to substantial overheads. To eliminate this overhead, HOBBIT implements two different optimizations. First, HOBBIT applies a class-sensitive optimization to restrict its protection to classes that are essential to the COOP attack. Second, HOBBIT applies the idea of MAC algorithm parameter randomization, thereby increasing overall security. For many application contexts, HOBBIT is thus the only viable defense against COOP.

Our contributions are as follows:

- We present HOBBIT, a software-only defense that thwarts counterfeit-object-oriented programming (COOP, for short).
- We describe the implementation of a fully-fledged Clang/LLVM-based prototype that supports all C++ features, such as multiple inheritance (see Sections 5 and 6).
- We discuss two new HOBBIT optimization techniques that enable users to balance their security needs with the available performance budget. We introduce Gadget-directed optimization (see Sections 5.5 and 7.5), to apply protections specifically to COOP gadgets, and Class-Hierarchy-driven Seed Randomization (see Sections 5.3 and 6.4).
- We evaluate HOBBIT w.r.t. performance, scalability, and security (see Section 7). Specifically, we report:
 - *Performance*: A configurable performance impact between 121.63% and 2.80%.
 - *Scalability*: HOBBIT compiles complex real-world software, such as the WebKit web browser.
 - *Security*: HOBBIT provides comprehensive security through either strong hashing techniques or randomizing parameters of weaker hashing techniques.

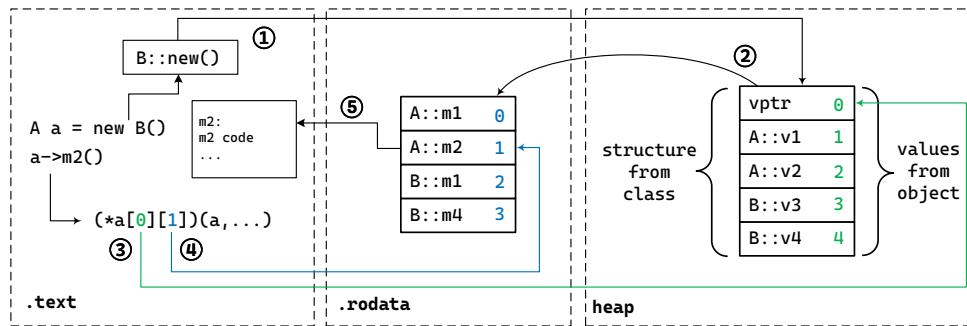


Figure 1 Overview of polymorphism and dynamic binding in C++. (1) constructors allocate objects and set `vptr` and field values, (2). Method calls require resolving of the `vtable`, (3), and then the corresponding fixed method id, (4), before being able to call the method, (5).

2 Background

In this section, we will introduce the background needed to understand the Counterfeit Object-Oriented Programming (COOP) attack. Since COOP is a high-level attack targeting specific C++ semantics, we will briefly explain the C++ object layout, polymorphism, and dynamic dispatch mechanism. Finally, we need to cover some preliminary concepts used in HOBBIT.

2.1 C++ Polymorphism and Dynamic Binding

In object-oriented programming languages, such as C++, programs are organized around classes and objects. *Classes* in C++ define the fields of objects and the *methods* operating upon them. A concrete instance of a class is called *object* and consists of values for the defined data fields residing in a contiguous memory region. To create and initialize newly created objects, programmers call special methods, so-called *constructors*.

Figure 1 illustrates these concepts. Instantiating a new B object triggers a call to its constructor (1), which allocates a contiguous memory region and sets the `vptr` due to the concrete dynamic type (2). The class determines the structure of each object, while the object holds values specific to the instance.

To dynamically bind a method call, C++ uses so-called `vtable`-based method dispatch (see Figure 1). For each class, C++ generates a corresponding `vtable` that holds the addresses of each callable method on it. If a class inherits a method, its address will merely be copied into the corresponding method slot. If a class overrides a method, a new address will be written into the corresponding method slot. A method call, then, consists of two steps: (i) resolving the `vtable` by dereferencing an object and accessing the first entry, which holds the `vtable` reference (3), and (ii) resolving the method by dereferencing the proper method through a callsite-fixed method identifier (4).

2.2 Counterfeit-Object-Oriented Programming (COOP)

Over the past four decades, the memory unsafe nature of C and C++ lead to an “Eternal War in Memory” [51]. In the beginning, attackers were able to insert instructions as data in writable memory. By facilitating a buffer overflow to overwrite the return address,

attackers could hijack the control-flow of a program to execute injected code, resulting in Arbitrary Code Execution (ACE). Simple defenses, such as Write exclusive-or Execute (W^X) – marking memory as either writable or executable, but not both – render such code injection attacks impossible. Therefore, attackers adapted and began reusing existing code, residing in executable memory. Attackers either reused whole functions (e.g., return-into-libc [21, 35]) or performed arbitrary computations by chaining together small pieces of code, called *gadgets*, as in Return-Oriented Programming (ROP) [46, 42, 49] and its variations [10, 19, 15, 44]. Many defenses targeting mentioned Code Reuse Attacks (CRAs) exist [28, 37, 1, 2, 11]. A more recent CRA targeting high-level C++ semantics is COOP [45].

COOP exploits the dynamic dispatch mechanism and escapes mentioned defenses above. Instead of introducing new invalid control-flows like in ROP or return-into-libc, COOP misuses existing callsites. To illustrate this point, consider the example from the previous section, but from the perspective of the CPU. A callsite merely fixes the method identifier, but accepts *any vtable* and will, thus, invoke *any* method identified by the fixed method identifier (see Figure 1, ⑤.)

COOP abuses this property of *vtable*-based method dispatch, by injecting malicious objects, so-called *counterfeit objects*. These objects use invalid *vtable* entries, to abuse method invocation. Instead of abusing gadgets as in return-oriented programming, COOP abuses whole functions. Since the notion of code-reuse attacks is tied to the nomenclature of *gadgets*, COOP, too, defines whole-function reuse gadgets.

These COOP gadgets are methods that can be abused for a specific malicious purpose. Not all COOP gadgets are equally important, though. The most important gadget is the so-called *main-loop gadget*, or ML-G for short. Consider the following C++ method:

```

1  virtual void removeElement(Element x) {
2      for (int i= 0; i < this.N; i++) {
3          this.L[i].remove(x);
4      }
5  }
```

 **Listing 1** Example of a COOP main-loop gadget (ML-G).

As shown in Listing 1, the `removeElement` method will loop over an array, namely the field `L` and invoke the virtual method `remove` on every object stored in the field `L`. From an adversarial COOP perspective, this means that the attacker can inject arbitrary malicious objects and store them in the corresponding `L` field. Once she can invoke the `removeElement` method, the attack will be launched.

More advanced variants of COOP relax this requirement for a container object holding references to other objects. Crane et al., for example, describe *Recursive-* and *Unrolled COOP* variants that allow different patterns of repetition [20]. By applying control-flow integrity, valid control-flow transfers can be restricted to the program’s call-graph. Chen et al. demonstrates that COOP can still succeed despite this constrain [16]

2.3 Execute-Only Memory (XOM)

Machine code in the `text` section of a program usually possesses read *and* execute privileges. The read privilege is required to process inlined data, such as jump tables for `switch` statements. But the read privilege requirement is not *strict*. The only essential privilege for code is the ability to execute. Inlined data must then move to another section with read privileges.

The principle of least privilege – a core tenet of computer security policies – prescribes that reducing privileges improves security. Thus, in the 60s the Multics project already supported execute-only memory [18]. Over the past decade, the idea of execute-only memory saw a revival [4, 47, 19, 20, 24, 6]. The revival was due to advanced, sophisticated multi-stage attacks that used memory leaks to (i) read a processes code layout, and then to (ii) relocate a generic attack to the specific code layout used by a program. These specific code layouts were derived from an active research area called “software diversity,” and complementing existing methods with execute-only memory begot the new class of defenses called *leakage-resilient diversity*.

2.4 Message Integrity Through MACs

To verify the authenticity and integrity of a message sent over an untrusted medium, people use so-called *message-authentication codes*, MACs for short. Both sender and receiver agree on a message authentication code (MAC) algorithm, based on a shared secret key k . Then, the sender computes the MAC checksum, also known as tag t , for every message m : $t = MAC(m, k)$ and sends this tag t along with the message. At the receiving end, we recompute the tag t' for the received message m' : $t' = MAC(m', k)$. Then, by comparing both tags t and t' for equality, we verify the message m 's integrity. Since the MAC algorithm is based on a secret key k , only shared between sender and receiver, third-parties cannot compute valid tags. Typically, secure MAC algorithms are based on cryptographic keyed-hash functions.

Counterfeit-object-oriented programming exploits the fact that control data, such as `vptrs` are mixed with non-control data. Similar to buffer overflows, mixing both types of data proves to be a security problem when adversaries inject malicious objects.

3 Related Work

Due to the severity of counterfeit-object-oriented programming as an attack vector, a variety of defenses [26, 20, 40, 57, 5, 16] has been proposed. Prior work, thus, considers multiple different design criteria. These design criteria include: software-only [20, 5] vs hardware-based [31, 54], hardening applied to binaries [41, 56, 23, 22] vs software-only, differences w.r.t. protected program parts (such as, protecting `vtables`, `vtable`-pointers, or dynamic dispatch). Due to these differences, giving an exhaustive treatment is in direct conflict within traditional scope restrictions. We therefore focus on the most directly related work, and skip, e.g., prior work dealing with securing C++ programs without source code access.

Most closely related to HOBBIT is CFIXX, which uses Intel's discontinued MPX extension to protect `vptrs` [12]. At its core, CFIXX separates `vptrs` from `vtables` and stores them into a dedicated memory area protected by MPX. In 2022, Xie et al. demonstrated a CFIXX version building on Intel's Control-Flow Enforcement Technology (CET) [54]. Recently, many defenses proposed the use of Intel's MPK extension. Unfortunately, using MPK is not compositional: If an application uses MPK itself, it cannot share its MPK use with any other component, such as a defense.

Compared to HOBBIT, CFIXX highlights the need for a software-only approach that does not require specific hardware extensions beyond extended-page table support to enable execute-only memory.

CCFI, short for cryptographically-secured control-flow integrity, is another closely related defense – not specifically aimed at preventing COOP attacks, but providing comprehensive protection against essentially all forms of control-flow hijacking [31]. CCFI pioneers the use of MACs to protect code pointers. Unfortunately, to secure the keys from leaking, the system proposed to reserve vector registers (i.e., SSE’s `xmm` registers), thus slowing down application relying on their use, such as media en- or decoders.

Compared to HOBBIT, CCFI highlights the need to preserve performance characteristics of programs, primarily by finding alternatives to protect secret keys that do not result in prohibitive performance impacts.

Hardware-based approaches are inextricably bound to the hardware mechanism and thus prone to sun-setting, as in the case of Intel’s MPX instructions, or lack of compositionality, as in the case of MPK extensions.

Defenses based on cryptographic primitives often suffer from poor performance, e.g., by effectively blocking vector registers, and the security-prerequisite of having cryptographic primitives not spill data onto the stack.

4 Threat Model

COOP is a rather sophisticated attack and will, thus, often be a last resort for attackers. We assume, consequently, that proper defenses against simpler attacks, such as code injection, ROP [46, 42], and return-into-libc [21] are in place. Since HOBBIT aims to prevent COOP attacks, we assume a strong threat model in line with previous work [45, 20, 31, 12].

In general, launching a COOP attack requires an attacker to hijack an initial object, including its virtual table pointers (`vptrs`) and data, and inject new counterfeit objects. To that end, an attacker needs to read or infer addresses of Virtual Tables (`vtables`) and write object-like data, including `vptrs` and other data, to specific memory regions. A variety of vulnerabilities provide such capabilities, including buffer overflows [38] and use-after-free vulnerabilities [51]. Although a restricted read- and write capability might suffice, we assume an attacker capable of reading arbitrary readable memory and writing to arbitrary writable memory.

Our system relies on `W^X`, marking memory either as writable or executable, but not both at the same time. Writing to code residing in executable memory or execute written data is not possible. Therefore, injecting new code or modifying existing code is not possible.

The attacker’s arbitrary read capability renders defenses relying on secrets in readable memory ineffective. For example, protecting against overflowing into control data, such as return addresses or `vptrs` with (stack) canaries, is not effective. An attacker can easily read these values and embed them in her payload, or – assuming an arbitrary write capability – skip canaries at all. To mitigate this issue, we assume Execute-Only Memory (XOM), therefore, we consider values or functions in XOM as secret.

Finally, we assume an attacker with specific knowledge about the target program and system. First, he has access to the target program’s source code. Second, she is able to infer the base address of the initial object, and the addresses of virtual function gadgets (`vfgadgets`) located in C++ modules. Although COOP relies primarily on high-level C++ semantics, some `vfgadgets` rely on specific instructions or registers, e.g., `vfgadgets` for loading argument registers to pass arguments to other `vfgadgets`. An attacker requires at least partial knowledge of the binary layout to use some `vfgadgets`. Third, the attacker knows about the system’s configuration, including deployed defenses, software versions, and hardware features.

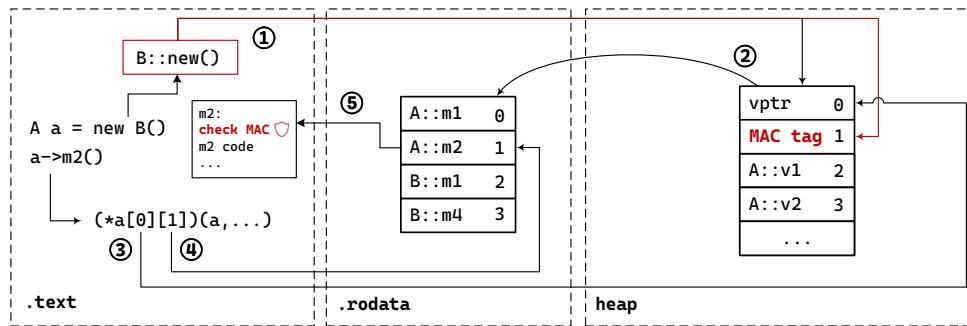


Figure 2 HOBBIT changes to C++. (1) constructors allocate objects, set `vptr` and field values and compute a MAC value, (2). Method calls are resolved as before, see Figure 1, but all method prologs now validate the MAC value, (5).

5 Design Aspects of Hobbit

HOBBIT is, broadly speaking, a defense that monitors and validates integrity. Whenever this integrity is violated by an adversary, we know that the program is under attack. A direct consequence of any integrity-protection mechanism also holds for HOBBIT: we protect neither the injection, nor the modification of objects; subsequent method calls trying to *act* on maliciously-modified objects will detect integrity violations.

The integrity monitored by HOBBIT is the object to `vptr` binding. One could just add a random value into an object and repeatedly validate its value. Since our threat model includes a powerful attacker with memory read capabilities, choosing a simple random value is insecure. Instead, HOBBIT considers objects, more specifically `vptrs`, between constructors and methods as messages, and secures them by applying message-authentication codes.

The following sections provide an in-depth discussion of the relevant design aspects of HOBBIT. Section 5.1 discusses C++ relevant aspects of object lifetime and changing the object layout to add the MAC tag. Sections 5.2 and 5.3 describe the benefits of using execute-only memory, and MAC-algorithm diversification. Section 5.4 lists possible locations for verifying signatures. Finally, we introduce the concept of gadget-directed optimization in Section 5.5.

5.1 C++ Object Lifetime and Layout

Objects in C++ live between construction and destruction, i.e., by constructors and destructors, respectively. Constructors instantiate an object by initializing, or assigning concrete values to its fields, which themselves are prescribed by their corresponding class definitions. Since a `vptr` is merely a field itself, at least from a run-time perspective, the constructor assigns the `vptr` of the called dynamic type. Destructors clean up object instances and, finally, free the allocated memory.

HOBBIT changes the C++ object layout by adding a machine-word per `vptr` that holds the computed MAC tag (see Figure 2 (1)). Besides requiring an extra word per `vptr`, such a change breaks the application binary interface (ABI), and we discuss the implications thereof in Section 6.

5.2 Message-Authentication Codes and Execute-Only Memory

In HOBBIT, we consider vptrs as messages sent from constructors (see Figure 2 (1)) and received by virtual methods (see Figure 2 (5)). The key security aspect of MAC functions is the shared-secret key between senders and receivers. If an attacker retrieves this secret key, she can craft valid signatures for malicious messages, thus violating the authenticity property of sent messages. To prevent leakage of this shared-secret key, HOBBIT piggybacks on execute-only memory’s leakage-resilience property.

Execute-only memory means that the adversary is precluded from reading code memory. As a result, we can hide privileged information directly in code memory. HOBBIT hides two privileged pieces of information there: (i) keys as intermediate constants, and (ii) MAC algorithm implementations. Hiding implementations from adversaries forces them to guess, thus further frustrating attacks.

MAC algorithm parameters, too, are important for security. Consider the following parameterization to compute object-vptr tags:

$$t = \text{MAC}(\text{vptr} \oplus r) \quad (1)$$

Although we include a random parameter r to the MAC computation, our attacker can use their memory-read primitive to read an object – including its `vptr` and the corresponding tags – and, use it later on during an attack at a different location. Such a staged attack is called a “replay” attack. To counter these replay attacks, we need to add the `vptr` location to the computation:

$$t = \text{MAC}(\text{vptr} \oplus \&\text{vptr} \oplus r) \quad (2)$$

By making MAC tags location-dependent, the attacker cannot trivially replay the object layout she read at a different location.

Prior defenses reserve registers to hold the key and exclude them from register allocation [31, 39]. Since the compiler then never allocates these registers, the key stored therein is considered safe from attackers. Although simple, this solution suffers from two drawbacks. First, reserving registers increases register pressure, which is particularly problematic on architectures with few registers, such as x86. Second, whether a key stored in registers is actually safe, depends on additional measures and precautions for context switches. Through its use of execute-only memory, HOBBIT bypasses these shortcomings.

5.3 Class-Hierarchy-Driven Seed Randomization

By using just a single random parameter r in our MAC tag computation, the adversary can bypass HOBBIT, once he identifies both the secret MAC algorithm and the value of r . HOBBIT counters this problem by using as many random parameters r as possible. In theory, different random parameters r can be randomly assigned across an application. In practice, however, we need to preserve C++ semantics across type-compatible call-sites. A conservative way to ensure semantics preservation is to map a single random parameter r to a subgraph in the class hierarchy graph (see Section 6.4). A more aggressive way would be to factor in run-time information, e.g., through profiling.

Due to this additional security mechanism, we can also loosen the strength requirements for our MAC algorithm. By choosing small, but efficient pseudo hash functions, such as `moremurmur-hash` [32], HOBBIT users favor performance over security, and vice versa. Since MAC algorithm implementations are protected by execute-only memory (see Section 2.3), the perceptible loss of security is minimal.

HOBBIT supports a wide variety of MAC hashing algorithms, such as `blake3`, `highwayhash`, `xxhashct`, `moremurmur`, and `moremurmur-random`.

5.4 Validating MAC Tags

HOBBIT recomputes and validates tags stored in objects in function prologs of virtual functions (see Figure 2, (5)). Although an attacker can inject malicious objects, HOBBIT will detect tampering with a tag *after* resolving the dynamic type, but *before* executing the actual method body. Alternatively, HOBBIT can also validate tags already at virtual call sites, but this implies embedding MAC hash computation into every call site, thus increasing the amount of machine instructions for each call site. Depending on the chosen MAC function implementation (e.g., inlined), these additional machine instructions might result in a considerable binary size increase.

In C++, most compilers use `vptrs` for other run-time related features besides dynamic dispatch. The use of run-time type information (RTTI), for example, requires loading the `rtti` pointer from the `vtable`. Similarly, dynamic casts use information stored in `vtables`, such as offsets to access/identify sub-objects for multiple inheritance. Although HOBBIT could validate tags in these cases, too, we choose to focus protection on dynamic dispatch, which is the key objective for COOP attacks.

5.5 Gadget-Directed Optimization

For performance-critical systems, such as real-time applications, HOBBIT can relax security and optimize for speed. Since COOP relies on special gadgets for dispatching other gadgets, we can embed integrity checks only in methods acting as such gadgets. To prevent attackers from executing Main Loop Virtual Function Gadgets (ML-Gs), HOBBIT can perform static analysis on source code to identify methods iterating over a collection of objects and calling virtual functions on them (see Section 6.5.)

HOBBIT could also analyze binaries to identify gadgets relying on binary instructions. Muntean et al., for example, created a tool for identifying gadgets and automating a COOP attack [34]. In general, identifying all gadgets is difficult and since variants of COOP exist, the resulting defense may not be complete [20, 16].

6 Hobbit Implementation

We implemented our prototype of HOBBIT as compile-time transformations on top of LLVM/Clang 17.0.3 [17] for the `x86_64` Linux platform and Itanium ABI [25]. Most researchers implement their prototypes as passes in LLVM that operate on and modify the LLVM specific intermediate representation, short LLVM IR. However, we implemented most parts of HOBBIT in Clang, since compilation is a lossy transformation and high-level C++ information, e.g., virtual methods and their callsites, are not – at least without complex analysis – available in LLVM IR.

First, HOBBIT extends the object layout to reserve space for the newly introduced MAC tag fields. After reserving space for MAC tags, we add instructions for computing and storing MAC tags in objects to constructors. For the final part of the `vptr` validation, we implement MAC tag checks in virtual methods. In Section 6.3 we describe our different MAC function implementations, Section 6.4 shows HOBBIT’s diversification implementation, and Section 6.5 demonstrates a prototype of our Gadget-directed Optimization. Section 6.6 lists the limitations of our prototype implementation of HOBBIT.

6.1 Extending Object Layouts

Extending the object layout requires us to change the size of objects in a special data structure called `RecordLayouts`. Clang uses the type `CXXRecordDecl` to represent C++ structs, unions, and classes. `RecordLayouts` store information about fields, their offsets, paddings, and lengths, (virtual) bases, and other layout-related information. Since HOBBIT introduces a new MAC tag field, we have to increase the size of the layout accordingly. On `x64` systems, pointers are eight byte long. Therefore, we add eight bytes to the (data-) layout size for dynamic `CXXRecordDecls` that do not inherit `vptr` (and consequently the MAC tag field) from a parent class in `AST/RecordLayoutBuilder` (`ItaniumRecordLayoutBuilder::LayoutNonVirtualBases`). Later, during the lowering of records, we add the field information for our MAC tag field, right after the `vptr` (see Listing 2).

```

1 void CGRecordLowering::accumulateVPtrs() {
2     if (Layout.hasOwnVFPtr()) {
3         auto vfptra = ...;
4         Members.push_back(vfptra);
5         auto HobbitMACField = MemberInfo(getSize(vfptra.Data),
6                                         MemberInfo::Field,
7                                         getIntNTType(64));
8         Members.push_back(HobbitMACField);
9     }
10    ...
11 }
```

Listing 2 Add MAC tag field while lowering records.

Extending the object layout breaks the C++ ABI compatibility. By recompiling the entire toolchain, including a standard C++ library, we still can compile and run programs with our C++ ABI modifications. We encountered one error in the `libunwind` library regarding macro definitions for the size of `libunwind::UnwindCursor`. `libunwind::UnwindCursor` is a dynamic class, therefore, consists of a `vptr` and with HOBBIT also a MAC tag field. To fix this error we have to account for the new tag field and thus add one to all macro definitions defining the constant `_LIBUNWIND_CURSOR_SIZE` in `__libunwind_config.h`. With this simple fix, HOBBIT can compile even the largest C++ programs.

6.2 Computing and Validating MAC Tags

C++ programs adhering to the C++ standard create objects solely by calling constructors. Therefore, we decided to implement the MAC tag computation and storing of the results in constructors. Constructors already perform the `vptr` initialization in a function called `CodeGenFunction::InitializeVTablePointer`. Likewise, HOBBIT initializes the MAC tags right after `vptr` initialization. Listing 3 shows the resulting assembly code of a constructor compiled with HOBBIT. A standard `clang` compiler emits the three assembly instructions (lines 3–5) initializing the `vptr` of an object of a class `B`. Since `_ZTV1B` points to the beginning of the `vtable` – the first two entries in the `vtable` are the offset-to-top and the RTTI pointer – the compiler adds 16 bytes to the `vtable` such that the `vptr` points to the first virtual function and finally saves the `vptr` in the designated field at the beginning of the given object. The remaining instructions (lines 7–12) are emitted by HOBBIT and responsible for loading

```

1 _ZN1BC2Ev:
2 ...
3 leaq    _ZTV1B(%rip), %rcx # load address of vtable
4 addq    $16, %rcx          # add 2 qwords for 1st virt. function = vptr
5 movq    %rcx, (%rax)      # store vptr at beginning of object
6 # HOBBIT START #
7 movq    (%rax), %rdx      # load vptr into rdx register
8 movq    %rax, %rcx         # load this into rdx register
9 xorq    %rdx, %rcx         # vptr xor this
10 movabsq $random, %rdx     # load secret value r to rdx
11 xorq    %rdx, %rcx         # xor secret value r = mac tag
12 movq    %rcx, 8(%rax)      # save mac tag to designated field
13 # Possible inlined hashing or call to compiler-rt hash function
14 # HOBBIT END #
15 ...

```

Listing 3 x86_64 assembly for an exemplary constructor of a dynamic class B emitted by HOBBIT.

```

1 _ZN1A2m2Ev:
2 # start function prolog:
3   # save callee-saved registers
4   # set up stack for local variables
5   #
6 # HOBBIT START #
7 movq    (%rcx), %rdx      # load vptr to rdx
8 movq    %rcx, %rax         # load this ptr to rax
9 xorq    %rdx, %rax         # vptr xor this
10 movabsq $random, %rdx     # load secret value r to rdx
11 xorq    %rdx, %rax         # xor secret value r = mac tag'
12 movq    8(%rcx), %rcx      # load saved mac tag
13 cmpq    %rcx, %rax         # check if tag' = tag
14 jne     .LBB4_2            # on mismatch jump to trap
15 ... # actual function      # actual function body
16 .LBB4_2: # %MACMismatchBlock # block with trap for mac tag mismatch
17 movl    $147, %edi          # store result code 147 to edi
18 callq   exit@PLT           # exit(147) on mac tag mismatch
19 # HOBBIT END #

```

Listing 4 x86_64 assembly for an exemplary virtual method of a dynamic class B emitted by HOBBIT.

both `vptr` and `this` in registers, followed by the `xor` instruction. The `movabsq` instruction loads an immediate – the random secret r – to a register and `xor` it to the previous result. Finally, the `xor` result is written to the MAC tag field, 8 bytes after the `vptr`.

HOBBIT inserts MAC tag validation checks in virtual functions (see Listing 4). These validation checks protect against attackers calling virtual functions on objects with fake or altered `vptrs`, therefore mitigating COOP attacks. If HOBBIT should protect dynamic

■ **Table 1** Details of implemented MAC functions used for benchmarking.

Name	MAC Function	Implementation
baseline	—	—
no-hash	none; only <code>xor(vptr, &vptr, random_secret)</code>	—
blake3	C implementation of BLAKE3	static lib
highwayhash	highwayhash	shared lib
xxhashct	compile-time implementation of xxhash	static lib
moremur	pseudo hash function based on moremur	inlined
moremur-random	diversified version (random parameter) of moremur	inlined

casts or RTTI access, we could insert MAC validation checks at those locations as well. To prevent the execution from virtual function bodies HOBBIT inserts the following instructions in `CodeGenFunction::StartFunction`:

1. We retrieve all `vptrs` for the current object.
2. For each `vptr`, we compute the MAC tag again.
3. For each `vptr`, we load the stored MAC tag value.
4. Then, we compare the computed and loaded MAC tag values.
5. If these tags match, we start executing the function body.
6. Otherwise, we detect an ongoing COOP attack and can launch counter-measures. In our prototype implementation, we simply exit the program with status 147.

Listings 3 and 4 show the resulting assembly code for both constructors and virtual methods of a class with 1 `vptr` without any hashing (`no-hash`).

6.3 MAC Function Implementations

We implemented different MAC functions in HOBBIT and extended the `baseline`, an unmodified Clang/LLVM 17.0.3. Table 1 shows the different hash implementations for the MAC function. The simplest approach is `no-hash` (as shown in Listings 3 and 4) that uses the identity function as MAC in Equation (2). Therefore, tag t is the unhashed result of the `xor` operations.

In contrast, `moremur` [32] implements a pseudo-hash function as MAC. These pseudo hash functions should be small, such that HOBBIT inlines these hash functions in both constructors and virtual functions. With XOM, immediate values used in such hash functions are resistant to leakage and can thus be considered secret. Section 6.4 describe `moremur-random`, a diversified implementation variant of `moremur`.

```

1   ...                                # preceding instructions from Listing 3
2   movabsq $random, %rax             # load random value to rax
3   xorq    %rax, %rdi               # xor random value = mac tag
4   callq   coop_hash@PLT            # call to compiler-rt hash function
5   movq    %rax, %rcx               # store result of coop_hash to rcx
6   movq    -16(%rbp), %rax          # reload this pointer
7   movq    %rcx, 8(%rax)            # save mac tag to designated field
8   ...

```

■ **Listing 5** Constructor calling a hash function in `rt-lib`.

We implemented the remaining MAC functions, all including larger and more complex hash functions, as compiler run-time libraries, short `compiler-rt`. LLVM provides and links these libraries for run-time support in compiled binaries. We implemented different versions of such a `compiler-rt` for the remaining MAC variants `blake3` [9], `highwayhash` [3], and `xxhashct` [55]. HOBBIT links the `compiler-rt` libraries for `blake3` and `xxhashct` statically to the program under compilation. `Highwayhash`, in contrast, is dynamically linked as a shared library.

With run-time hashing support enabled, HOBBIT simply inserts a call to the hash function located in the run-time library, according to Equation (2). Listing 5 shows the resulting instructions. After the initial `xor` instructions, the result is passed as an argument to the `coop_hash` function. The function `coop_hash` computes a hash according to the chosen hash function (Table 1), namely `blake3`, `highwayhash`, or `xxhashct`. Finally, after loading the `this` pointer again, the returned result is stored in the designed MAC tag field.

6.4 Class-Hierarchy-Driven Seed Randomization

In its current implementation, the random parameter r of Equation (2) is fixed over the whole program. We implemented a naive diversification approach diversifying this random parameter. Ideally, we would choose a different parameter for each class, however, due to the polymorphic nature of C++, the diversification degree is limited. We create MAC tags in constructors and validate them in virtual functions, therefore, both MAC functions must use the same random parameter. With subtyping, methods must be callable for different classes, according to the inheritance graph. Therefore, our diversified implementation chooses a random parameter for each weakly connected subcomponent of the inheritance graph. The inheritance graph is, in fact, a directed acyclic graph¹, since C++ has the concept of multi-inheritance, hence the famous *diamond problem*.

We implemented `moremurmur-random` in the following steps:

1. In an initial compilation step, HOBBIT outputs all classnames with the corresponding (virtual-) bases.
2. We implemented a Python script that constructs the inheritance DAG.
3. Our script assigns each weak component² a different random parameter r .
4. HOBBIT then use this class assignment to diversify the MAC tag computation.

By enabling `link-time optimization`, we could implement the inheritance graph analysis and the diversification assignment in Clang/LLVM.

6.5 Gadget-Directed Optimization

We implemented a simple gadget-directed optimization that identifies simple main-loop gadgets. With this optimization enabled, HOBBIT performs a static analysis to identify potential main-loop gadgets. Our naive analysis checks whether a virtual method belongs to a class declaring any fields of C++ standard container type [13], either directly or indirectly, by inheriting from classes with such fields. This prototype gadget-directed optimization only identifies simple main-loop gadgets, but fails to identify other forms of dispatcher gadgets, serving as a main-loop gadget [45, 20]. Other dispatcher gadgets include `recursive gadgets`, `unrolled COOP`, or iterators over linked lists. HOBBIT could use COOP exploit automation frameworks, such as `iTOP`, to identify additional gadgets and feed them into our gadget-directed optimization [34].

¹ Not a tree, as one would expect.

² All connected subgraphs, also called *components*, ignoring the direction of edges.

 **Table 2** Benchmark system configuration.

	EPYC 7H12	i7-8559U	Ryzen 9 5900X
Processor	AMD EPYC 7H12	Intel 8559U	AMD Ryzen 9 5900X
RAM	1 TB DDR4	64 GB DDR4	64 GB DDR4
OS	Debian 12	Debian 12	Ubuntu 22.04.4 LTS
Kernel	6.1.0-16-amd64	6.1.0-16-amd64	6.5.0-27-generic
gcc	12.2.0	12.2.0	11.4.0
glibc	2.36	2.36	2.35
linker	gold (2.38)	gold (2.38)	GNU ld (2.38)

6.6 Limitations

HOBBIT does not protect RTTI objects. RTTI objects are dynamic types, but not created by calling constructors at run-time. Instead, Clang initializes RTTI objects during compilation, therefore, HOBBIT does not compute and store MAC tags for such objects. At load-time, vtables and RTTI objects alike are loaded into `.rodata`. However, protecting RTTI objects is still possible but requires extra effort. We could, for example, create initialization code similar to our MAC tag initialization in constructors and call this RTTI object initializer when the address of both vtables and RTTI objects is known, at load-time. Since HOBBIT does not create MAC tags for RTTI objects, we do not emit integrity checks in virtual functions belonging to RTTI classes.

7 Evaluation

We present the evaluation of our prototype implementation of HOBBIT. In Section 7.1, we describe the machines used for our evaluation. Sections 7.2–7.4 show the performance evaluation, including measurements of run-time, memory-usage, and code-size. We evaluate our implemented prototype of gadget-directed optimization in Section 7.5. In Section 7.6, we evaluate the scalability of HOBBIT by compiling real-world applications with HOBBIT. Finally, Section 7.7 shows the evaluation of the class-hierarchy-driven seed randomization.

7.1 System Configuration

We perform our evaluation of HOBBIT on three different machines listed in Table 2.

We used machines EPYC 7H12 and i7-8559U for the performance evaluation in Section 7.2 and the gadget-directed optimization evaluation in Section 7.5. The scalability evaluation in Section 7.6 and the evaluation of the diversification statistics in Section 7.7 were done on Ryzen 9 5900X.

Our prototype of HOBBIT is based on the LLVM/Clang version 17.0.3 (see Section 6), which we call **baseline** in the following evaluation. Since HOBBIT breaks the C++ ABI, we have to build and use a custom-built version of the LLVM C++ standard library `libc++` [29] (same as LLVM/Clang: 17.0.3). To improve comparability – although not strictly necessary – we build and use a custom-built `libc++` for the baseline as well.

7.2 Performance

As common in performance evaluations, we evaluate the performance of HOBBIT by building the SPEC CPU 2017 benchmark with our compiler modifications. In particular, since HOBBIT only applies changes to C++ programs, we run the four C++ benchmarks of the SPECSpeed™

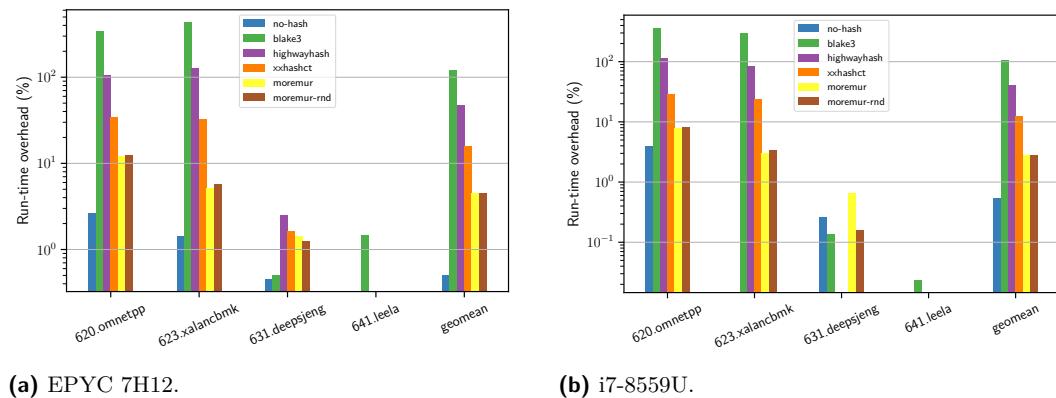


Figure 3 Run-time overhead introduced by HOBBIT for C++ benchmarks of the SPECspeed™ 2017 Integer test suite, relative to baseline on log-scale.

2017 Integer test suite, namely 620.omnetpp, 623.xalancbmk, 631.deepsjeng, and 641.leela. The remaining non-C++ benchmarks showed – as expected – no measurable overhead. As mentioned in Section 7.1, we use the custom-built libc++ instead of the bundled version of the Linux distribution. Each experiment compiles all relevant benchmarks and runs the compiled benchmark afterwards. We repeated each experiment 10× on EPYC 7H12 and 6× on i7-8559U and calculated the geometric mean over those repetitions.

Run-time, a key metric within SPEC, quantifies the time in seconds required for a benchmark to execute. Figure 3 shows the results for all evaluated MAC functions (see Table 1).

For the i7-8559U machine, the geometric mean overhead over all benchmarks, are 107.62% (**blake3**), 40.40% (**highwayhash**), 12.21% (**xxhashct**), 2.83% (**moremur**), and 2.80% (**moremur-random**). In comparison, on EPYC 7H12, the benchmarks show a higher performance impact over all benchmarks, namely 121.63% (**blake3**), 47.81% (**highwayhash**), 16.02% (**xxhashct**), 4.49% (**moremur**), and 4.54% (**moremur-random**). Both, 620.omnetpp and 623.xalancbmk, show the most performance impact on both machines. On i7-8559U, 620.omnetpp shows the highest run-time increase consistently for all benchmarked MAC functions. In contrast, on EPYC 7H12, we see a significantly higher run-time overhead on 623.xalancbmk for **blake3** and **highwayhash** compared to 620.omnetpp. The remaining hash functions (**xxhashct**, **moremur**, and **moremur-random**) on EPYC 7H12 show the same trend as on i7-8559U, namely, a higher performance overhead for 620.omnetpp rather than 623.xalancbmk.

We also evaluated a stripped down version that does not compute MAC tags to measure the minimum overhead (**no-hash** in Figure 3). On i7-8559U **no-hash** introduces a geometric mean overhead of 0.55%, with a maximum performance impact of 4.00% (620.omnetpp). In contrast to “correct” hash functions, the implementation of **no-hash** is 7.27% faster on EPYC 7H12 (overall 0.51%; 620.omnetpp 2.66%) when compared to i7-8559U.

7.3 Memory

Since HOBBIT extends object layouts, therefore, increases the size of objects, we are interested in the maximum **resident set size** (RSS). RSS is a metric indicating the memory usage of a process in RAM. Swapped memory does not count to RSS. By querying the **rusage** counters [43], our benchmarking environment measures the maximum RSS **maxrss**.

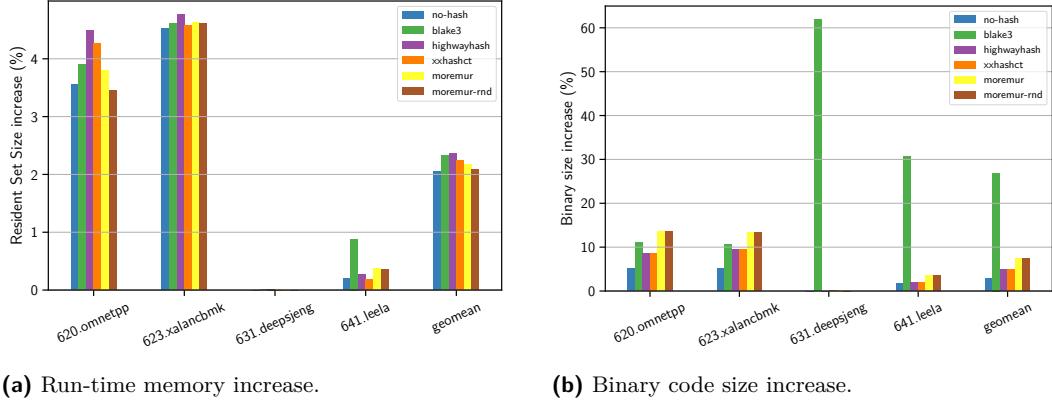


Figure 4 Memory effects of HOBBIT for C++ benchmarks of the SPECspeed™ 2017 Integer suite, relative to baseline (EPYC 7H12).

Table 3 Binary sizes of benchmarks and run-time libraries for both machines EPYC 7H12 and i7-8559U.

(a) Binary sizes of baseline benchmarks.

Name	Size in Bytes
620.omnetpp	2,915,320
623.xalancbmk	7,362,408
631.deepsjeng	118,120
641.leela	254,936

(b) Binary sizes of hashing run-time libraries.

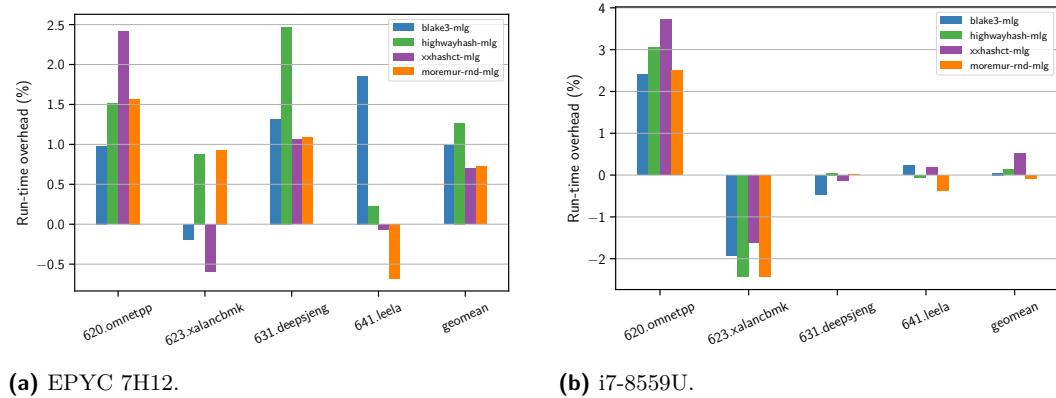
Name	Size in Bytes
blake3	90,618
highwayhash	15,816
xxhashct	1,874

Figure 4 shows the benchmarking results for machines EPYC 7H12 and i7-8559U. On both machines, our benchmarks show an overall geometric `maxrss` overhead of 2.2% and 2.18%, respectively. We see the highest `maxrss` overhead for 623.xalancbmk (EPYC 7H12 4.64%, i7-8559U 4.65%). 620.omnetpp has a similar `maxrss` overhead (EPYC 7H12 3.99%, i7-8559U 3.88%), whereas HOBBIT has a low `maxrss` impact on 641.leela (EPYC 7H12 0.41%, i7-8559U 0.32%). For 631.deepsjeng, our defense does not increase the `maxrss` on neither machine at all.

7.4 Code Size

HOBBIT inserts instructions for creating and validating MAC tags and, for some MAC functions, links run-time libraries and creates function calls to these libraries. These additional instructions (and libraries) increase the binary size of compiled programs. To that end, we evaluate the binary size of each benchmark. Table 3 shows the binary sizes of the baseline benchmarks (see Table 3a) and the run-time hashing libraries (see Table 3b).

The binary size increase on both machines is identical and shown in Figure 4b. HOBBIT, in its `blake3` variant, introduces the highest geometric mean increase in binary size of 26.90% over all benchmarks, ranging from 10.50% for 623.xalancbmk up to 61.87% for 631.deepsjeng. `Blake3` is a big hashing library (see Table 3b). Since HOBBIT links `blake3` statically to the compiled program, the big library size, compared to small benchmarks as in 631.deepsjeng and 641.leela, contributes to the significant increase in the resulting hardened binary. On the other hand, for `highwayhash`, nearly 8.5× bigger than `xxhashct`, accounts for roughly



(a) EPYC 7H12.

(b) i7-8559U.

Figure 5 Reduction of performance impact through gadget-directed optimization.

the same binary size increase as `xxhashct`. The reason for this similar increase in binary size – despite a different library size itself – results from a different linkage. `Highwayhash` is dynamically linked, whereas `blake3` and `xxhashct` are statically linked, therefore, embedded in the binary. HOBBIT variants that inline MAC functions in constructors and virtual functions, namely `mormeur` and `moremurm-random`, introduce the highest increase in binary size for 620.omnetpp (13.54%) and 623.xalancbmk (13.36%).

7.5 Gadget-Directed Optimization

We evaluated our naive implementation for the main-loop gadget analysis optimization (see Section 6.5), that only creates and validates MAC tags for classes having a standard C++ container field.

Although our gadget-directed optimization finds no main-loop gadgets for benchmarks 623.xalancbmk, 631.deepsjeng, and 641.leela, it finds 12 instances of classes having – directly or indirectly – at least one container-type field. HOBBIT inserts MAC tag integrity validation logic in 137 methods of these 12 classes.

Figure 5 shows the run-time overhead introduced by HOBBIT with gadget-directed optimization enabled.

7.6 Scalability

To evaluate the scalability of HOBBIT, we compiled WebKit, a web browser engine consisting of millions of lines of C and C++ code (see Table 5). Specifically, we built the GTK version of Webkit, `WebKitGTK` [53], a full-featured port of WebKit for GTK-based Linux desktop systems. Although HOBBIT breaks the C++ ABI through its object-layout extension, we only needed a single change to successfully compile WebKit, shown in Listing 6. Since `ScrollableArea` is a dynamic class, HOBBIT inserts a field for the MAC tag, thus we have to add 8 to this `static_assert` to account for the increased object size.

After the compilation, we evaluated the run-time overhead introduced by our defenses with the following browser benchmarks: (i) Kraken, (ii) MotionMark, (iii) Octane, and (iv) Speedometer.

As this evaluation requires a GUI, we performed the experiments on Ryzen 9 5900X. With only a terminal window opened, we started the `MiniBrowser`, a minimal browser based on `WebKitGTK`. After each benchmark execution, we closed the `MiniBrowser`, waited for ten

```

1  #if CPU(ADDRESS64)
2  -static_assert(sizeof(ScrollableArea) == sizeof(
3      SameSizeAsScrollableArea),
4      "ScrollableArea should stay small");
5  +static_assert(sizeof(ScrollableArea) == sizeof(
6      SameSizeAsScrollableArea) + 8,
7      "ScrollableArea should stay small");
8  #endif

```

■ Listing 6 Fix required to compile WebKitGTK.

■ Table 4 Performance impact on browser benchmarks.

Benchmark	blake3	highwayhash	xxhashct	moremurmur-random
Kraken 1.1 [27]	2.72%	0.70%	0.77%	-2.05%
MotionMark 1.3 [33]	14.64%	1.67%	1.87%	3.03%
Octane 2.0 [36]	53.54%	17.32%	2.83%	1.34%
Speedometer 2.1 [50]	161.74%	43.24%	7.85%	2.54%

seconds and repeated the experiment. In total, we executed each benchmark three times. Table 4 shows the geometric mean performance impact of our evaluation. Kraken measures the time needed to finish the benchmark, therefore, an induced overhead means an *increase* in run-time. In contrast, the other benchmarks measure *score points*, meaning that an induced overhead *decreases* the achieved score.

These real-world benchmark results confirm the results obtained from compute-intensive programs. HOBBIT allows balancing security and performance, and we did not notice perceptible delays in daily browsing activities.

To further show that HOBBIT scales to other real-world programs, we successfully compiled the following programs listed in Table 5. We included the version of the compiled programs as well as their C++ source lines of code (SLOC). The selected programs range from small web frameworks to fully fledged web browsers and compiler. For measuring SLOCs we used the tool `sloccount` [48].

■ Table 5 Source lines of code (SLOC) of real-world programs compiled with HOBBIT.

Program	Description	Version	SLOC (C++)
crow	C++ Web framework	1.2.0	25,203
json	JSON library for C++	3.11.3	102,977
llvm	Collection of compiler tools	17.0.6	2,201,374
webkitgtk	GTK port of WebKit	2.41.1	4,444,590
620.omnetpp	SPECspeed@2017 Integer suite	SPEC CPU 2017	63,100
623.xalancbmk	SPECspeed@2017 Integer suite	SPEC CPU 2017	243,046
631.deepsjeng	SPECspeed@2017 Integer suite	SPEC CPU 2017	7,284
641.leela	SPECspeed@2017 Integer suite	SPEC CPU 2017	30,473

 **Table 6** Top-5 and overall weakly connected component set size for libc++, C++ benchmarks of SPECspeed™ 2017 Integer, and WebKitGTK.

Top 5	libc++	omnetpp	xalanc	deepsjeng	leela	WebKitGTK
1.	78	193	442	78	78	3,916
2.	45	78	93	45	45	1,962
3.	27	52	78	27	27	1,541
4.	13	45	62	13	14	1,066
5.	12	34	49	12	13	427
Overall	197	379	539	197	252	30,438

7.7 Class-Hierarchy-Driven Seed Randomization

We evaluated the number of diversified random parameters for our implementation from Section 6.4. Table 6 shows the Top-5 weakly connected components, that constitute the diversification unit. Each of these units is a set of classes for whom we must choose the same random parameter. All C++ benchmarks of SPECspeed™ 2017 Integer and WebKitGTK depend on libc++ and, thus, include and extends libc++’s inheritance graph. 631.deepsjeng does not introduce any new dynamic classes to the inheritance graph, whereas WebKitGTK adds 30,241 new weakly connected components.

8 Discussion

We discuss and interpret the relevant findings of our evaluation.

8.1 Performance

Our performance evaluation shows that the performance impact depends primarily on the choice of the MAC algorithm. Although `blake3` offers the highest security, its performance impact, too, is the highest. To improve performance, HOBBIT offers two complementary options. First, users can opt to use simpler MAC algorithms, such as `moremur`, which is more performance friendly. Second, users can apply our gadget-directed optimization to reduce performance impact of even the most expensive MAC algorithms.

Since we did not find any impact on large, real-world software, such as the WebKit browser, we argue that HOBBIT can be used in a wide variety of contexts.

8.2 Security

We compiled the `CFIxx-Suite` [14] with our HOBBIT compiler. This exploit coverage test suite, created by Burow et al., demonstrates several scenarios for attacks on the dynamic dispatch mechanism [12]. Our security evaluation of HOBBIT is shown in Table 7. HOBBIT, in its initial version, only protects against scenarios 3–5 (namely `VTxchg`, `VTxchg-hier`, `COOP`), but fails to detect scenarios 1–2 (namely `FakeVT`, `FakeVT-sig`).

The initial HOBBIT implementation prevents malicious execution of virtual function bodies by validating the integrity of `vptrs` in the function prologue. Since scenarios 1–2 insert fake `vtables` that contain pointers to non-virtual functions, therefore unprotected by our defense, our prototype implementation does not prevent this form of attacks.

Table 7 Results of testing different `vtable` related attack building blocks against LLVM, LLVM CFI, and different configurations of HOBBIT.

Exploit	LLVM	LLVM-CFI	Hobbit	Hobbit+LLVM-CFI	Hobbit-VFCS
FakeVT	✗	✓	✗	✓	✓
FakeVT-sig	✗	✓	✗	✓	✓
VTxchg	✗	✓	✓	✓	✓
VTxchg-hier	✗	✗	✓	✓	✓
COOP	✗	✗	✓	✓	✓

However, HOBBIT is compatible and composable with other defenses such as LLVM CFI [30]. We compiled the exploit coverage test suite with HOBBIT again, this time with `vcall sanitizer` enabled. To enable LLVM CFI, we provided the following compiler flags:

```
-fsanitize=cfi-vcall -fno-sanitize-recover=vcall -fno-sanitize=vptr
```

LLVM CFI succeeds in defending against fake `vtable` attacks and limits successful virtual calls to valid subtypes of the dispatched object’s static type. Still, LLVM CFI fails to prevent an attacker from maliciously changing `vptrs` adhering to the type hierarchy or inserting fake objects without calling the appropriate constructor – the core principle of COOP. Combining HOBBIT with LLVM CFI protects against all five exploit types evaluated in the exploit coverage test suite.

To account for situations where CFI cannot be used, we implemented an extension of HOBBIT, namely HOBBIT-VFCS. This HOBBIT extension moves validation code from the function prologue of virtual functions to their call sites. HOBBIT-VFCS validates `vptrs` *after* loading the `vptr` (Figure 2 ③), but *before* invoking the method call (Figure 2 ④). Emitting validation checks at each call site increases the binary size, but mitigates all five exploits. In future work, we can apply the same principle – checking the validity of `vptrs` immediately after loading – to protect other `vtable` related mechanisms, such as dynamic casts, too.

8.2.1 Balancing Performance and Security

HOBBIT has, essentially, two orthogonal compile-time parameters: (i) hash function algorithm selection, and (ii) validation code granularity. By selecting a strong hash function, such as `blake3`, the overall security improves at the cost of performance. Conversely, selecting a more efficient hash function, such as `moremurmur`, decreases security and increases performance.

To offset the performance penalties, HOBBIT offers users to parameterize the granularity of validation code insertion. Either all virtual functions or only COOP-relevant call sites are protected. By protecting all call sites, HOBBIT achieves the highest security at the potentially highest performance impact (i.e., by selecting an “expensive” hash function). Conversely, by selecting only the COOP-relevant call sites, HOBBIT reduces performance impact to a negligible level.

Although four different levels can be specified, we recommend the following settings in practice. A strong hash function, such as `blake3`, should be combined with COOP-relevant gadget granularity. A weak hash function, such as `moremurmur`, can be used to protect all virtual functions.

8.2.2 Uniformly Distributed Vtables

A method to perform cryptanalysis is to correlate input with output characteristics. Known-plaintext attacks are a form where the attacker knows the plaintext and infers a model from the outputs. In our model both inputs and outputs are either known or can be read directly through a memory-read primitive. The MAC algorithm used is hidden away effectively through execute-only memory. Yet, some of the input characteristics may allow attackers to launch a known-plaintext attack.

Consider, for example, that the attacker knows the addresses of `vtables` v_1 , v_2 , and v_3 . Let's assume that although the addresses of these `vtables` v_i are different, their distances may remain constant. An adversary could, therefore, rely on such constant inter-table differences to infer properties about the concrete hash MAC algorithm used by HOBBIT.

Although our present implementation does *not* address this issue, we can achieve uniform distribution of inter-table differences by way of randomizing the order of emitting `vtables`. If this randomization proves to be insufficient, padding entries can be added in between emitted `vtables` to increase the entropy of `vtable` addresses.

9 Conclusions

HOBBIT presents an integrity-protection mechanism to thwart counterfeit-object-oriented programming attacks. At its core, this attack shares a symmetry to classical buffer overflows, in the sense that the underlying problem is the mixing of control with non-control data. For buffer overflows, this mix consists of keeping return addresses among stack frame data. For COOP attacks, this mix consists of keeping the `vptr` among object field data. By injecting malicious objects, the adversary can thus hijack control-flow and initiate illegitimate method calls.

To stop this type of whole-function code-reuse attack, HOBBIT changes the object layout to embed a tag value. This tag is computed by MAC functions that encode `vptr` information, `vptr` location, and a random secret. By leveraging execute-only memory, HOBBIT provides additional security. Due to complementary optimizations, users gain the ability to balance performance and security.

A comprehensive analysis provides evidence of both (i) configurable performance impact between 121.63% and 2.80% and (ii) scalability to multi-million lines of C and C++ code. At the same time, HOBBIT does not depend on MPX and does not inhibit performance by reserving registers. Without any hardware requirements, HOBBIT is applicable to embedded- and IOT devices.

References

- 1 Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In Vijay Atluri, Catherine Meadows, and Ari Juels, editors, *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA, November 7-11, 2005*, pages 340–353, New York, New York, USA, April 2005. ACM. doi: 10.1145/1102120.1102165.
- 2 Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):4:1–4:40, November 2009. doi:10.1145/1609956.1609960.
- 3 Jyrki Alakuijala, Bill Cox, and Jan Wassenberg. Fast keyed hash/pseudo-random function using SIMD multiply and permute. *CoRR*, abs/1612.06257, December 2016. doi:10.48550/arXiv.1612.06257.

- 4 Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberg, and Jannik Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 1342–1353, New York, New York, USA, 2014. ACM. doi:10.1145/2660267.2660378.
- 5 Markus Bauer and Christian Rossow. Novt: Eliminating C++ virtual calls to mitigate vtable hijacking. In *IEEE European Symposium on Security and Privacy, EuroS&P 2021, Vienna, Austria, September 6-10, 2021*, pages 650–666. IEEE, September 2021. doi:10.1109/EuroSP51992.2021.00049.
- 6 Felix Berlakovich and Stefan Brunthaler. R2C: aocr-resilient diversity with reactive and reflective camouflage. In Giuseppe Antonio Di Luna, Leonardo Querzoni, Alexandra Fedorova, and Dushyanth Narayanan, editors, *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*, pages 488–504, New York, NY, USA, May 2023. ACM. doi:10.1145/3552326.3587439.
- 7 Matthias Bernad. HOBBIT implementation. Software (visited on 2024-08-29). URL: <https://github.com/mbernad/hobbit-artifact>.
- 8 Matthias Bernad and Stefan Brunthaler. HOBBIT. Software (visited on 2024-08-29). URL: <https://doi.org/10.5281/zenodo.11046716>.
- 9 BLAKE3/c at master · BLAKE3-team/BLAKE3. URL: <https://github.com/BLAKE3-team/BLAKE3/tree/master/c>.
- 10 Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In Bruce S. N. Cheung, Lucas Chi Kwong Hui, Ravi S. Sandhu, and Duncan S. Wong, editors, *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2011, Hong Kong, China, March 22-24, 2011*, pages 30–40, New York, New York, USA, 2011. ACM. doi:10.1145/1966913.1966919.
- 11 Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Comput. Surv.*, 50(1):16:1–16:33, April 2017. doi:10.1145/3054924.
- 12 Nathan Burow, Derrick Paul McKee, Scott A. Carr, and Mathias Payer. CFIXX: object type integrity for C++. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, Reston, VA, February 2018. The Internet Society. doi:10.14722/ndss.2018.23279.
- 13 C++ Containers. URL: <https://cplusplus.com/reference/stl/>.
- 14 CFIXX Suite. URL: <https://github.com/HexHive/CFIXX/tree/master/CFIXX-Suite>.
- 15 Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pages 559–572, New York, New York, USA, 2010. ACM. doi:10.1145/1866307.1866370.
- 16 Kaixiang Chen, Chao Zhang, Tingting Yin, Xingman Chen, and Lei Zhao. Vscape: Assessing and escaping virtual call protections. In Michael D. Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 1719–1736. USENIX Association, August 2021. URL: <https://www.usenix.org/conference/usenixsecurity21/presentation/chen-kaixiang>.
- 17 Release LLVM 17.0.3 · llvm/llvm-project. URL: <https://github.com/llvm/llvm-project/releases/tag/llvmorg-17.0.3>.
- 18 Fernando J. Corbató and Victor A. Vyssotsky. Introduction and overview of the multics system. In Robert W. Rector, editor, *Proceedings of the 1965 fall joint computer conference, part I, AFIPS 1965 (Fall, part I), Las Vegas, Nevada, USA, November 30 - December 1, 1965*, pages 185–196. ACM, November 1965. doi:10.1145/1463891.1463912.

- 19 Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical code randomization resilient to memory disclosure. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, volume 2015-July, pages 763–780. IEEE Computer Society, May 2015. doi:10.1109/SP.2015.52.
- 20 Stephen J. Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. It's a trap: Table randomization and protection against function-reuse attacks. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 243–255, New York, New York, USA, 2015. ACM. doi:10.1145/2810103.2813682.
- 21 Solar Designer. lpr LIBC RETURN exploit, August 1997. URL: <https://insecure.org/splights/linux.libc.return.lpr.sploit.html>.
- 22 Mohamed Elsabagh, Dan Fleck, and Angelos Stavrou. Strict virtual call integrity checking for C++ binaries. In Ramesh Karri, Ozgur Sinanoglu, Ahmad-Reza Sadeghi, and Xun Yi, editors, *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*, pages 140–154, New York, NY, USA, April 2017. ACM. doi:10.1145/3052973.3052976.
- 23 Robert Gawlik and Thorsten Holz. Towards automated integrity protection of C++ virtual function tables in binary programs. In Charles N. Payne Jr., Adam Hahn, Kevin R. B. Butler, and Micah Sherr, editors, *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014*, pages 396–405, New York, New York, USA, 2014. ACM. doi:10.1145/2664243.2664249.
- 24 Jason Gionta, William Enck, and Peng Ning. Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In Jaehong Park and Anna Cinzia Squicciarini, editors, *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, CODASPY 2015, San Antonio, TX, USA, March 2-4, 2015*, pages 325–336. ACM, March 2015. doi:10.1145/2699026.2699107.
- 25 Itanium C++ ABI. URL: <https://itanium-cxx-abi.github.io/cxx-abi/abi.html>.
- 26 Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Safedispatch: Securing C++ virtual calls from memory corruption attacks. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014. doi:10.14722/ndss.2014.23287.
- 27 Kraken JavaScript Benchmark (version 1.1). URL: <https://mozilla.github.io/krakenbenchmark.mozilla.org/index.html>.
- 28 Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. Sok: Automated software diversity. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 276–291. IEEE Computer Society, May 2014. doi:10.1109/SP.2014.25.
- 29 “libc++” C++ Standard Library – libc++ documentation. URL: <https://libcxx.llvm.org/>.
- 30 LLVM: Control Flow Integrity. URL: <https://clang.llvm.org/docs/ControlFlowIntegrity.html>.
- 31 Ali José Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: cryptographically enforced control flow integrity. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, volume 2015-October, pages 941–951. ACM, October 2015. doi:10.1145/2810103.2813676.
- 32 Mostly mangling: Stronger, better, morer, Moremur; a better Murmur3-type mixer. URL: <https://mostlymangling.blogspot.com/2019/12/stronger-better-morer-moremur-better.html>.
- 33 MotionMark 1.0. URL: <https://browserbench.org/MotionMark/>.

- 34 Paul Muntean, Richard Viehoever, Zhiqiang Lin, Gang Tan, Jens Grossklags, and Claudia Eckert. itop: Automating counterfeit object-oriented programming attacks. In Leyla Bilge and Tudor Dumitras, editors, *RAID '21: 24th International Symposium on Research in Attacks, Intrusions and Defenses, San Sebastian, Spain, October 6-8, 2021*, pages 162–176. ACM, October 2021. doi:10.1145/3471621.3471847.
- 35 Nergal. Advanced return-into-lib(c) exploits (PaX case study), December 2001. URL: <http://phrack.org/issues/58/4.html#article>.
- 36 Octane 2.0 JavaScript Benchmark. URL: <https://chromium.github.io/octane/>.
- 37 Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-free: defeating return-oriented programming through gadget-less binaries. In Carrie Gates, Michael Franz, and John P. McDermott, editors, *Twenty-Sixth Annual Computer Security Applications Conference, ACSAC 2010, Austin, Texas, USA, 6-10 December 2010*, pages 49–58, New York, New York, USA, 2010. ACM. doi:10.1145/1920261.1920269.
- 38 Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.
- 39 Taemin Park, Julian Lettner, Yeoul Na, Stijn Volckaert, and Michael Franz. Bytecode corruption attacks are real - and how to defend against them. In Cristiano Giuffrida, Sébastien Bardin, and Gregory Blanc, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 15th International Conference, DIMVA 2018, Saclay, France, June 28-29, 2018, Proceedings*, volume 10885 of *Lecture Notes in Computer Science*, pages 326–348. Springer, 2018. doi:10.1007/978-3-319-93411-2_15.
- 40 Andre Pawłowski, Victor van der Veen, Dennis Andriesse, Erik van der Kouwe, Thorsten Holz, Cristiano Giuffrida, and Herbert Bos. VPS: excavating high-level C++ constructs from low-level binaries to protect dynamic dispatching. In David M. Balenson, editor, *Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC 2019, San Juan, PR, USA, December 09-13, 2019*, pages 97–112, New York, NY, USA, December 2019. ACM. doi:10.1145/3359789.3359797.
- 41 Aravind Prakash, Xunchao Hu, and Heng Yin. vfguard: Strict protection for virtual function calls in COTS C++ binaries. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, Reston, VA, November 2015. The Internet Society. doi:10.14722/ndss.2015.23297.
- 42 Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, March 2012. doi:10.1145/2133375.2133377.
- 43 getrusage(2) - Linux manual page. URL: <https://man7.org/linux/man-pages/man2/getrusage.2.html>.
- 44 AliAkbar Sadeghi, Salman Niksefat, and Maryam Rostamipour. Pure-call oriented programming (PCOP): chaining the gadgets using call instructions. *J. Comput. Virol. Hacking Tech.*, 14(2):139–156, May 2018. doi:10.1007/s11416-017-0299-1.
- 45 Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, volume 2015-July, pages 745–762. IEEE Computer Society, May 2015. doi:10.1109/SP.2015.51.
- 46 Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, pages 552–561, New York, New York, USA, 2007. ACM. doi:10.1145/1315245.1315313.
- 47 Zhuojia Shen, Komail Dharsee, and John Criswell. Fast execute-only memory for embedded systems. In *IEEE Secure Development, SecDev 2020, Atlanta, GA, USA, September 28-30, 2020*, pages 7–14. IEEE, September 2020. doi:10.1109/SecDev45635.2020.00017.
- 48 SLOCCount. URL: <https://dwheeler.com/sloccount/>.

- 49 Kevin Z. Snow, Fabian Monroe, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 574–588. IEEE Computer Society, May 2013. doi:10.1109/SP.2013.45.
- 50 Speedometer 2.1. URL: <https://browserbench.org/Speedometer2.1/>.
- 51 Laszlo Szekeres, Mathias Payer, Tao Wei, and R. Sekar. Eternal war in memory. *IEEE Secur. Priv.*, 12(3):45–53, May 2014. doi:10.1109/MSP.2014.44.
- 52 Arjan Van De Ven. New security enhancements in red hat enterprise linux v.3, update 3, 2004. URL: https://static.redhat.com/legacy/f/pdf/rhel/WHP0006US_Execshield.pdf.
- 53 WebKit/WebKit at webkitgtk-2.41.1. URL: <https://github.com/WebKit/WebKit/tree/webkitgtk-2.41.1>.
- 54 Mengyao Xie, Chenggang Wu, Yinqian Zhang, Jiali Xu, Yuanming Lai, Yan Kang, Wei Wang, and Zhe Wang. CETIS: retrofitting intel CET for generic and efficient intra-process memory isolation. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, CCS ’22, pages 2989–3002, New York, NY, USA, 2022. ACM. doi:10.1145/3548606.3559344.
- 55 ekpyron/xxhashct: Compile time implementation of the 64-bit xxhash algorithm as C++11 constexpr expression. URL: <https://github.com/ekpyron/xxhashct>.
- 56 Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. Vtint: Protecting virtual function tables’ integrity. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, pages 8–11, Reston, VA, 2015. The Internet Society. doi:10.14722/ndss.2015.23099.
- 57 Chao Zhang, Dawn Song, Scott A. Carr, Mathias Payer, Tongxin Li, Yu Ding, and Chengyu Song. Vtrust: Regaining trust on virtual calls. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, Reston, VA, 2016. The Internet Society. doi:10.14722/ndss.2016.23164.

Understanding Concurrency Bugs in Real-World Programs with Kotlin Coroutines

Bob Brockbernd 

Delft University of Technology, The Netherlands

Nikita Koval 

JetBrains, Amsterdam, The Netherlands

Arie van Deursen  

Delft University of Technology, The Netherlands

Burcu Kulahcioglu Ozkan  

Delft University of Technology, The Netherlands

Abstract

Kotlin language has recently become prominent for developing both Android and server-side applications. These programs are typically designed to be fast and responsive, with asynchrony and concurrency at their core. To enable developers to write asynchronous and concurrent code safely and concisely, Kotlin provides built-in *coroutines* support. However, developers unfamiliar with the coroutines concept may write programs with subtle concurrency bugs and face unexpected program behaviors. Besides the traditional concurrency bug patterns, such as data races and deadlocks, these bugs may exhibit patterns related to the coroutine semantics. Understanding these coroutine-specific bug patterns in real-world Kotlin applications is essential in avoiding common mistakes and writing correct programs.

In this paper, we present the first study of real-world concurrency bugs related to Kotlin coroutines. We examined 55 concurrency bug cases selected from 7 popular open-source repositories that use Kotlin coroutines, including IntelliJ IDEA, Firefox, and Ktor, and analyzed their bug characteristics and root causes. We identified common bug patterns related to asynchrony and Kotlin's coroutine semantics, presenting them with their root causes, misconceptions that led to the bugs, and strategies for their automated detection. Overall, this study provides insight into programming with Kotlin coroutines concurrency and its pitfalls, aiming to shed light on common bug patterns and foster further research and development of concurrency analysis tools for Kotlin programs.

2012 ACM Subject Classification Computing methodologies → Concurrent programming languages; Software and its engineering → Software testing and debugging

Keywords and phrases Kotlin, coroutines, concurrency, asynchrony, software bugs

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.8

Acknowledgements We thank our shepherd, Elisa Gonzalez Boix, and anonymous reviewers for their suggestions for improving the paper.

1 Introduction

Kotlin is a cutting-edge programming language that has recently become a primary language for Android development. Its modern syntax, seamless interoperability with Java, and enhanced features have positioned Kotlin as the preferred language for creating mobile and server-side applications for many developers. A standout feature amongst Kotlin's diverse functionalities is the built-in *coroutines* [14] support, which significantly simplifies asynchronous programming. Coroutines offers developers a streamlined approach to handling background tasks, thus enabling more intuitive and readable code.

Kotlin coroutines enables writing asynchronous code in a sequential style, thus avoiding the complexity of multithreaded programs. One may think of coroutines as lightweight threads. Following the notion of coroutines in the literature [12], Kotlin coroutines can

 © Bob Brockbernd, Nikita Koval, Arie van Deursen, and Burcu Kulahcioglu Ozkan;
licensed under Creative Commons License CC-BY 4.0

38th European Conference on Object-Oriented Programming (ECOOP 2024).

Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 8; pp. 8:1–8:20

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

suspend and resume their execution, and they do that through the *suspending* functions. Suspending functions (marked with the `suspend` keyword in Kotlin) are similar to *async* functions in other asynchronous languages and frameworks [37, 33], which encapsulate asynchronous computations but look like synchronous code, and calling them looks similar to calling regular functions. A suspending function indicates that it can be suspended in the middle of computation and resumed later. By suspending and resuming at predefined points, they yield computing resources to other coroutines and coordinate their execution. For example, when a suspending function encounters a long-running task and suspends, the underlying thread does not block but takes another coroutine to execute.

While coroutines simplify writing asynchronous programs and increase performance by asynchronously running long-running tasks, asynchronous programming comes with additional challenges. Besides the inherent concurrency non-determinism of multithreading, which causes traditional classes of concurrency bugs (such as data races, deadlocks, order violations, and atomicity violations), utilizing coroutine requires developers to reason about the asynchronous interactions between the concurrent components due to coroutine semantics, synchronous/asynchronous execution contexts, suspension and resumption of coroutines. Developers unfamiliar with the Kotlin coroutine semantics can use coroutines and suspending functions improperly, resulting in subtle concurrency bugs.

Kotlin is a relatively new language, so we know little about common misconceptions and bug patterns in programs with Kotlin coroutines. This work aims to shed light on them.

Our contribution. In this study, we explored real-world concurrency bugs in programs that use Kotlin coroutines. We collected concurrency bugs from popular open-source code repositories and identified common patterns in these bugs, their root causes, and misconceptions that might cause them. The results are available in the following GitHub repository [6].

Our analysis shows that some concurrency bugs are related to *bridging synchronous and asynchronous executions*. While asynchronous functions can only be called on a coroutine and the syntax of suspendable functions helps *function coloring* by marking asynchronous functions, composing synchronous parts of the program with asynchronous functions remains a challenge to programmers inexperienced in asynchronous programming. As a quick remedy for calling asynchronous functions from synchronous functions, they may call these functions in blocking coroutines, which in turn may affect the program's performance or even introduce deadlocks in case of *nested blocking calls*. On the other hand, when calling synchronous functions from asynchronous functions, they should be aware that the called function may run in an asynchronous context. Calling asynchronous functions from such functions can demand manual bookkeeping of coroutine scopes or omitting one of Kotlin's core concurrency principles: structured concurrency [34, 9]. In addition to *nested blocking calls*, we identified classes of bug patterns where the developers may violate structured concurrency by improper *scope passing*, *querying asynchronous objects*, *synchronization with asynchronous objects*, and improper *coroutine exception handling*.

In summary, our work makes the following contributions:

- We present the first (to the best of our knowledge) comprehensive study of real-world concurrency bugs in programs that use Kotlin coroutines, examining 55 concurrency bugs selected from 7 popular open-source repositories that use Kotlin coroutines.
- We identify common bug patterns related to Kotlin coroutines, presenting real-world examples and possible corrections for each, providing root causes and misconceptions that lead to these bugs, and discussing strategies for their automated detection.

Moreover, we communicated our findings to Kotlin developers. Our initial discussions show that our findings align with some of their observations, e.g. [29, 27]. We are in contact with them to develop inspection tools for identified bug patterns and have already contributed to IntelliJ IDEA with an inspection that detects one of the identified bug patterns [7].

Impact. We envision our findings contributing to developing reliable Kotlin programs targeting multiple audiences. They can help (i) Kotlin programmers better understand concurrency bugs and write correct programs, (ii) increase the awareness of programmers using Kotlin libraries about the potential asynchrony in the functions they use, (iii) provide insights to the Kotlin language team about possible misconceptions of programmers and (iv) researchers develop suitable concurrency analysis and testing tools for Kotlin programs.

2 Background on Kotlin Coroutines

Essentially, coroutines are lightweight threads that are relatively cheap to suspend and resume. They also support efficient cancellation, which, in sum, makes them very powerful for asynchronous programming. This section briefly introduces coroutines in Kotlin, discussing important differences compared to traditional threads and the features necessary to understand the bug patterns we present in our study.

Launching a coroutine. A coroutine is a computation unit not bound to any particular thread. Instead, coroutines run on threads and reuse them. When a coroutine gets suspended (pauses in the middle of computation), the underlying thread does not park but takes another coroutine and executes it, utilizing resources more efficiently. In this essence, coroutines can be considered a framework for managing expensive threads.

When launching a new coroutine, we can specify a coroutine dispatcher, which determines how to schedule the coroutine. The default dispatcher (`Dispatchers.Default`) is essentially a thread pool, where the number of threads is bound to the number of CPUs. It is also possible to launch coroutines on the `Main` dispatcher, ensuring that they are executed on the Main (UI) thread, or the `IO` dispatcher when the code does not compute something but blocks the running thread with an I/O operation.

Listing 1 shows an example with one coroutine launching another and printing “Hello”, and the second suspending for one second and printing “World!”. The `main()` function starts with a `runBlocking` call, which bridges the non-coroutine and coroutine worlds, blocking the current thread for the duration of the coroutine it runs. The `launch` call starts a new coroutine concurrently with the rest of the code on `Dispatchers.Default` coroutine dispatcher, which means this coroutine will run on a shared pool of threads. The `delay` call in the launched coroutine suspends it for one second. As result, this code prints “Hello” followed by “World!”.

Listing 1 Launching a new coroutine in Kotlin.

```

1 fun main() = runBlocking { // launches a coroutine on this thread
2     launch(Dispatchers.Default) { // launch a new coroutine
3         printWorldWithDelay()
4     }
5     println("Hello")
6 }
7

```

```

8 suspend fun printWorldWithDelay() {
9     delay(1000L) // non-blocking delay for 1 second
10    println("World!")
11 }

```

Suspending functions. To utilize coroutines, Kotlin provides the *suspending function* concept. Suspending functions, marked with `suspend` modifier, can be paused and resumed later without blocking the underlying thread. In Listing 1, `printWorldWithDelay()` and `delay(..)` are suspending functions, which might pause (in this case, the underlying thread switches to executing another coroutine). A suspending function can only be called from another suspending function, providing a structured way to write asynchronous and non-blocking code.

To summarize, the `suspend` keyword in Kotlin is used to mark a function that can be asynchronously completed – it can suspend its execution at some point, being resumed where it left off later, without blocking the underlying execution thread.

Structured concurrency. Coroutines follow a principle of structured concurrency, which means that new coroutines can only be launched in a specific scope, delimiting the lifetime of the coroutine. Structured concurrency ensures that they are not lost and do not leak. An outer scope cannot be completed until all its children's coroutines are complete, while cancellation of one of the coroutines in a scope instantly aborts the others within the scope.

In Listing 1, `runBlocking` launches a new coroutine and establishes a coroutine scope (accessible by `this` in the code block), so any coroutine launched within this block will cause this `runBlocking` call to wait until the launched coroutine finishes. This is why the `runBlocking` call does not complete until the second coroutine that prints “World!” finishes.

One may also specify a custom `CoroutineScope` to ensure that launched coroutines do not get lost and do not leak. Specifically, the scope finishes when all the coroutines launched within it are completed, while canceling the scope results in the cancellation of all the coroutines within it. Listing 2 contains the `printHelloWorld()` suspending function that, similarly to the code in Listing 1, launches a new coroutine and prints “Hello”, while the launched coroutine suspends for one second and prints “World!”. The coroutine scope here ensures that `printHelloWorld()` finishes only when the launched coroutine finishes. At the same time, in case the launched coroutine gets canceled, the whole scope gets canceled, resulting in `printHelloWorld()` cancellation.

■ Listing 2 Creating a custom `CoroutineScope` in Kotlin.

```

1 suspend fun printHelloWorld() = coroutineScope {
2     launch {
3         delay(1000L)
4         println("World!")
5     }
6     println("Hello")
7 }

```

Cancellation. Kotlin coroutines provide a built-in cancellation mechanism, which is especially useful for long-running applications. For example, a user might have closed the page that launched a coroutine, so its result is no longer needed, and the coroutine can be canceled. If a coroutine gets canceled while suspending, the respective `suspend` function throws `CancellationException` – the user code must not catch and always propagate it.

■ Listing 3 Coroutine communication via channel.

```

1 fun main() = runBlocking {
2     val channel = Channel<Int>(capacity = 1)
3     launch {
4         for (x in 1..5) channel.send(x * x) // sends to channel
5     }
6     repeat(5) {
7         println(channel.receive()) // receives from channel
8     }
9 }
```

With structured concurrency, when a coroutine is canceled, all the coroutines operating within the scope get canceled, too, thus ensuring that all the related computations are safely canceled and do not leak.

Channels. When programming with coroutines, developers typically use channels for implicit synchronization and communication instead of manipulating shared memory. A channel is a blocking queue of bounded capacity with `receive` operation suspending if the channel is empty and `send` suspending when the channel is full. Listing 3 illustrates an inter-coroutine communication via channel. One coroutine sends square numbers to the channel, and the main coroutine reads these numbers from the same channel and prints them.

Coroutines and threads. Coroutines do not introduce a new concurrency model but enable cooperative concurrency and efficient and safe thread management, with the structured concurrency feature and explicit communication primitives in particular. However, one may still program with coroutines in a way similar to programming with threads, sharing a mutable state (e.g., a concurrent cache) and using the same synchronization primitives (e.g., mutex).

Discussion. Programming with coroutines varies significantly from traditional thread-based programming. These differences might give rise to unique bugs distinct from those typically encountered when manipulating threads and shared memory. This work sheds light on popular concurrency bug patterns discovered in real-world applications with Kotlin coroutines.

3 Bug Study Methodology

Our bug study targets seven open-source repositories listed in Table 1 together with the numbers of Kotlin code lines, commits, and GitHub stars. The repositories are selected based on three criteria: (i) the repository mainly contains Kotlin code, (ii) the project depends on the Kotlin coroutines library, and (iii) the repository has a high number of commits, indicating its active development and high number of stars indicating its popularity and the interest of the community.

As Kotlin is the primary language for Android development, we started with identifying top-starred Android projects on GitHub and eliminated the projects that have less than 1,000 commits and lack descriptive commit messages. As a result, we obtained two repositories: the `Shadowsocks` proxy client [31] and the `Tachiyomi` comic reader [35]. Next, we determined the Android repositories with the highest commit count. After elimination, the

 **Table 1** The selected GitHub repositories for the bug analysis.

Repository	Commits	Stars	Kotlin LOC	Total LOC
JetBrains/intellij-community	427.4 K	16.1 K	1 603.4 K	9 805.0 K
wordpress-mobile/WordPress-Android	83.2 K	2.9 K	243.5 K	4 561.7 K
woocommerce/woocommerce-android	46.8 K	259	250.5 K	367.6 K
mozilla-mobile/firefox-android	30.2 K	1.3 K	431.1 K	717.0 K
jshaw29/tachiyomi-backup ¹	6.2 K	25.8 K	66.1 K	114.1 K
ktorio/ktor	5.1 K	11.8 K	152.5 K	152.6 K
shadowsocks/shadowsocks-android	3.6 K	34.3 K	7.8 K	12.2 K

following three were selected: the `WordPress` website builder [3], the `WooCommerce` webshop manager [2], and the `Firefox` web browser [25]. Additionally, we selected the `Ktor` [18] framework for building asynchronous server-side and client-side applications and `IntelliJ IDEA` Community Edition [17], both developed by JetBrains – the main maintainers of Kotlin coroutines.

To collect concurrency bugs related to Kotlin coroutines, we analyzed all commits in the selected repositories. Specifically, (1) we filtered the commits based on whether the commit messages contained specific concurrency-related keywords, and (2) manually reviewed the sifted commits. Finally, (3) we categorized the identified bugs by the root causes of the errors.

Filtering the commits based on the commit messages. Following prior research on concurrency bug studies [38, 40, 23], we selected the commits that include at least one of the following keywords: `race`, `deadlock`, `synchronization`, `concurrency`, `lock`, `mutex`, `atomic`, `compete`, or `semaphore`. With our work focusing on Kotlin coroutines, we expanded the filter with the coroutine-related keywords: `runBlocking`, `Dispatcher`, `CoroutineScope`, `cancel`, and `CancellationException`. For the feasibility of the manual analysis, we limited the number of selected commits associated with each keyword to the most recent 30 commits.

Manual analysis of the selected commits. After filtering the commits in the repositories, we had 1353 commits to analyze. We manually reviewed them, examining their commit messages and code changes. We selected the commits that fixed a concurrency bug involving Kotlin coroutine primitives, with the change being comprehensible without in-depth knowledge of the codebase. Thus, we also filtered out the classic concurrent bugs unrelated to coroutines. We ended up with 55 bugs that involve Kotlin coroutine constructs.

Manual analysis and categorization of the bugs. Finally, we categorized the filtered 55 bugs by their root causes, analyzing the programming patterns that led to the errors. The source code links to the studied bugs are available in our GitHub repository [6].

Classic concurrency bugs. As the goal of this work is to analyze concurrency bugs that are related to Kotlin coroutines, the study does not cover traditional multithreaded concurrency bug patterns [23] such as data races, order violations, or atomicity violations. Rather, we focus on the bug patterns related to and introduced by using Kotlin coroutines constructs.

¹ The Tachiyomi repository has been taken down since January 2024: <https://tachiyomi.org/news/2024-01-13-goodbye>, so we reference a backup repository instead.

Table 2 Collected bugs and their categorization into bug patterns of nested `runBlocking`, scope passing, querying asynchronous objects, synchronizing with cancellation, and `CancellationException` bugs. All observed bugs that did not fall into one of these categories are listed under “Uncategorized”.

Repository	Nested <code>runBlocking</code>	Scope Passing	Querying Async	Sync Cancel	Cancellation Exception	Uncategorized
IntelliJ	6	0	1	1	10	5
Firefox	2	2	2	2	0	4
Tachiyomi	2	0	0	0	3	1
Ktor	0	0	1	0	0	3
Shadowsocks	0	0	1	0	1	1
Wordpress	1	0	0	0	0	0
Woocommerce	0	2	0	1	0	3
Total	11	4	5	4	14	17

4 Categorization of Bugs

In this section, we analyze the bug patterns and root causes of the concurrency bugs and categorize them based on their characteristics. We discuss possible developer misunderstandings that lead to these mistakes, offer insight into possible remedial steps to rectify these errors and suggest methods for their automatic detection.

Table 2 lists the number of bugs in the repositories that fall into each of the bug categories. Based on our analysis of the Kotlin-specific concurrency primitives that are commonly involved in bugs and their root causes, we categorized the concurrency bugs in Kotlin programs into the classes of bugs due to (1) nested `runBlocking` calls (Section 4.1), (2) coroutine scope passing (Section 4.2), (3) querying asynchronous objects (Section 4.3), and (4) synchronizing with cancellation (Section 4.4). A special class of bugs occurs when a `CancellationException` is accidentally caught or incorrectly rethrown (Section 4.5). While it is not strictly a concurrency bug, it is caused by the Kotlin coroutines machinery, so we included it in our analysis.

Lastly, not all bugs found in the collection phase can be categorized into one of the categories. These bugs are displayed in Table 2 in the “Uncategorized” column. We omit these bugs in our analysis.

4.1 Calling `runBlocking` in a Coroutine

A common concurrency bug in Kotlin manifests when `runBlocking` coroutine builder is called from a coroutine and blocks the underlying coroutine dispatcher thread. Such a pattern can lead to a deadlock. We observed 11 bugs caused by this.

Root cause. The root cause of this bug is an improper use of the `runBlocking` coroutine builder designed to bridge non-coroutine and coroutine worlds and not expected to be called from another coroutine. Such an improper use can block the underlying scheduler thread, which might lead to a deadlock. Figure 1 and Listing 4 provide program examples with this bug pattern.

Figure 1 provides the code and the deadlock illustration. The program launches Coroutine *A* on the `Dispatcher.Main` dispatcher (line 2), dispatching the coroutine onto the UI thread. Then, coroutine *A* calls `nonSuspendingFunction()`. In turn, this function calls

`runBlocking` (line 8) which launches a coroutine *B* scheduled on `Dispatcher.Main` (line 9). The `runBlocking` builder blocks the UI thread and, due to structured concurrency, waits for coroutine *B* to finish. However, coroutine *B* cannot be dispatched until the UI thread is free. In other words, coroutine *A* blocks the thread that needs to execute coroutine *B*, while coroutine *A* also waits for coroutine *B* completion, which results in a deadlock.

The same deadlock might also occur in a multi-thread scenario. Listing 4 provides such an example, using the `Default` multi-threaded dispatcher to schedule coroutines. Similarly to the code in Figure 1, the program gets into a deadlock when all threads are executing the coroutines launched in `main()`, schedule new coroutines in `nonSuspendingFunction()` calls. However, these new coroutines cannot be executed – all the scheduler threads are occupied with the coroutines launched in `main()` and wait for the completion of these coroutines launched in `nonSuspendingFunction()`.

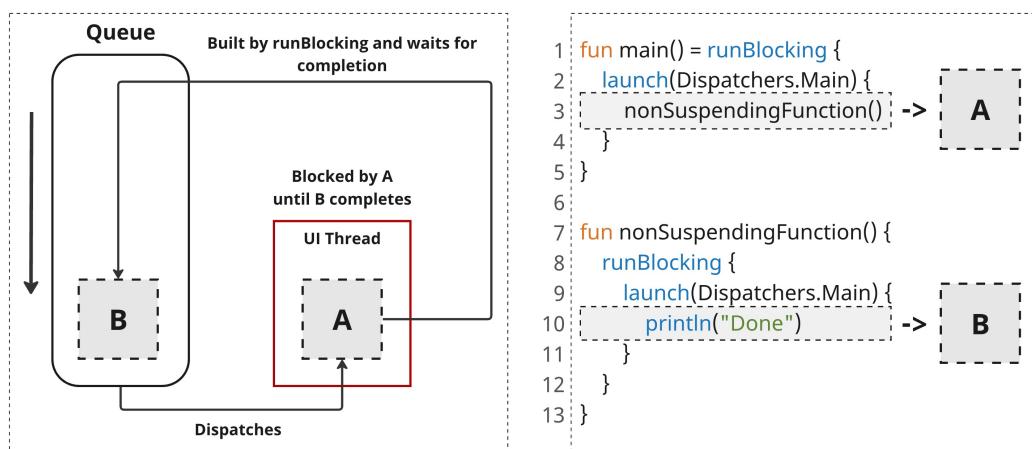


Figure 1 A program with a deadlock due to a `runBlocking` call from a coroutine on a single-threaded dispatcher.

Misconceptions. A common misconception is that the `runBlocking` builder can safely be used in non-suspending functions. However, a non-suspending function can be called from an asynchronous context; there is no guarantee that it runs outside a coroutine. Even if developers know they are working inside a coroutine, they might be unaware that the function they call contains a `runBlocking` builder. It is not always trivial to determine whether a piece of code runs inside a coroutine or whether a function call reaches a `runBlocking`, especially when the call stack is large.

When developers need to call a suspending function from a non-suspending function, they tend to call `runBlocking`, especially when the developer is unaware that this synchronous function actually runs in a coroutine. The right course of action, however, is not always clear and requires careful consideration by the developer. In the example program, turning `nonSuspendingFunction` into a `suspendingFunction` by adding the `suspend` keyword does the trick, as given as a potential solution in Listing 5. By turning the function into a suspend function, the developer can call `coroutineScope`, which allows for a normal `launch`. Note that this requires all functions calling `suspendingFunction` also to be suspending. In other cases, one might prefer to acquire a coroutine scope created elsewhere. This scope, however, comes with its own set of challenges, which we explain in Section 4.2.

Listing 4 A program with a deadlock due to a `runBlocking` call from a coroutine on a multithreaded dispatcher.

```

1 fun main() = runBlocking {
2     for(i in 1..1000) { // 1000 > max number of scheduler threads
3         launch(Dispatchers.Default) {
4             nonSuspendingFunction()
5         }
6     }
7 }
8
9 fun nonSuspendingFunction() {
10    runBlocking {
11        launch (Dispatchers.Default) {
12            println("Done")
13        }
14    }
15 }
```

Listing 5 A potential solution to the bug with nested `runBlocking` calls.

```

1 fun main() = runBlocking {
2     launch(Dispatchers.Main) { // launch coroutine A
3         suspendingFunction() // safe to call
4     }
5 }
6
7 suspend fun suspendingFunction() {
8     coroutineScope { // suspends execution until coroutine B is done
9         launch(Dispatchers.Main) { // launch coroutine B
10            println("Done")
11        }
12    }
13 }
```

Ultimately, both `runBlocking` and `coroutineScope` will pause the execution of the function calling it. The difference is that `runBlocking` does this by blocking the underlying thread and `coroutineScope` by suspending the coroutine, which releases the thread in the meantime.

A situation where it is nontrivial to identify parts of the codebase that run in coroutines and they may introduce unintended nested `runBlocking` calls might occur when the codebase is gradually migrated to use coroutines [10].

Possible automated detection. A static analysis can inspect the program's call graph and detect situations when `runBlocking` is called from a `suspend` function. This analysis could also be implemented as an IDE inspection, and we have successfully added one into IntelliJ IDEA [7].

4.2 Scope Passing

Scope passing is a coding pattern in which a function launches a coroutine in a coroutine scope created outside the function. This situation can lead to an unexpected execution order of program statements, which may violate their intended order. While passing the coroutine scope is not incorrect and sometimes might be necessary, it makes it difficult to reason about the execution order.

Root cause. The root cause is the nondeterminism in the completion time of the coroutines launched on an external scope. Listing 6 provides a program example for the bug pattern. Consider the function `loadTopPerformers` (line 4), which is responsible for loading some data and storing it in the `lastUpdateTopPerformers` variable (line 2). The developer expects the data to be available in the `lastUpdateTopPerformers` variable once the `loadTopPerformers` function returns. However, this is not necessarily the case: the coroutine scope used to launch the coroutine on line 5 is not bound by the function `loadTopPerformers`; it is inherited from the `DashboardViewModel` class.

As illustrated in Figure 2, the `this` object in the `loadTopPerformers` function refers to class `DashboardViewModel`, which extends `CoroutineScope`. This results in the coroutine unexpectedly outliving the function it was created in.

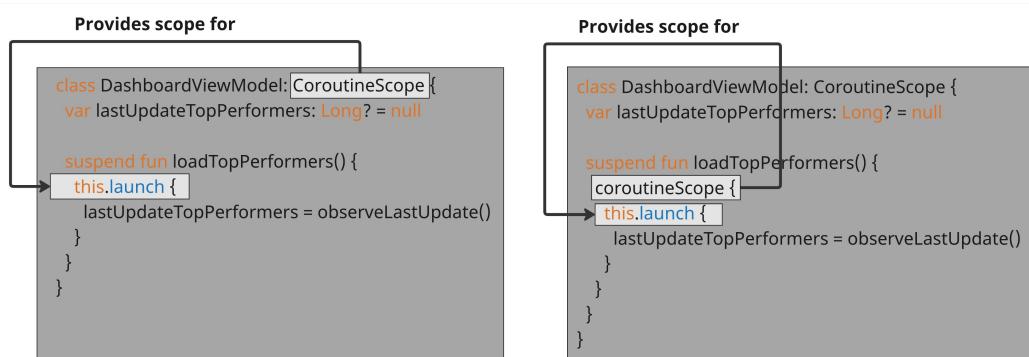


Figure 2 Visual clarification of coroutine scope origin and usage for Listing 6. Left shows the situation where the spawned coroutine can outlive the function it was created in. Right shows how this can be solved by creating a scope in the same function.

The problem can be fixed by ensuring that `loadTopPerformers` returns after the `lastUpdateTopPerformers` variable is set. As given in Listing 7, the example bug can be fixed by starting a `coroutineScope` call wrapping the function body (line 5). Then, the scope

Listing 6 An example scope passing bug, which is a simplified version of the bug in [32].

```

1 class DashboardViewModel: CoroutineScope {
2     var lastUpdateTopPerformers: Long? = null
3
4     suspend fun loadTopPerformers() {
5         launch {
6             lastUpdateTopPerformers = observeLastUpdate()
7         }
8     }
9 }

```

■ **Listing 7** A potential solution to example scope passing bug in Listing 6.

```

1 class DashboardViewModel: CoroutineScope {
2     var lastUpdateTopPerformers: Long? = null
3
4     suspend fun loadTopPerformers() {
5         coroutineScope {
6             launch {
7                 lastUpdateTopPerformers = observeLastUpdate()
8             }
9         }
10    }
11 }
```

■ **Listing 8** An example for a scope passing bug, which is a simplified version of the bug in [28].

```

1 class ProductShippingClassViewModel(): CoroutineScope {
2     private var loadJob: Job? = null
3
4     fun load() {
5         waitForCurrentLoadJob()
6         loadJob = launch { /* loading logic */ }
7     }
8
9     fun waitForCurrentLoadJob() {
10        launch { // launch since join cannot be called from normal fun
11            loadJob?.join() // join suspends until load job is done
12        }
13    }
14 }
```

suspends the function `loadTopPerformersStats` until all its children are completed. Note that this solution would not have been an option if `loadTopPerformers` was a non-suspending function since the `coroutineScope` can only be called from a suspending function.

Another example of a coroutine unexpectedly outliving its calling function due to launching on an external scope is provided in Listing 8. The function `load` starts an expensive load operation by spawning a coroutine (line 6). To ensure this operation only runs once at a time, the developer keeps a reference to its `Job` (line 2). Next, he creates a function `waitForCurrentLoadJob` that performs a `join` operation. Then, at line 5, this waiting function is called before the expensive load operation is started. However, since `waitForCurrentLoadJob` is a normal function, it cannot call the suspending `join` method. In order to access this `join` operation, a coroutine is launched (line 10). However, this coroutine is launched on a scope that is defined outside the `waitForCurrentJob` function. The wait function will, therefore, return immediately, allowing a new load job to be started before the old one is completed.

Misconceptions. A developer might expect a function that launches a coroutine to wait for the coroutine to finish since that is often the case due to structured concurrency. This, however, only holds when the coroutine scope is created in that same function. In the example of Listing 6, the `loadTopPerformersStats` sits in between scope creation and

coroutine launch. Listing 8 shows that a normal function `waitForCurrentLoadJob` calls a `join` operation by spawning a coroutine. As discussed in Section 4.1, a problem arises when a suspend function needs to be called from a non-suspending function that runs in a coroutine. The developer needs to choose between calling `runBlocking`, passing scope, or refactoring all depending code to suspend functions.

Automated detection. Detecting this bug pattern is challenging since there can be valid reasons for passing the scope. However, the problem is that the developer might be unaware that, in some cases, a launched coroutine outlives the function that launched it. A simple analysis can be made that checks whether the scope used to launch a coroutine is created in the same function or not. A simple and non-intrusive visual indication might aid the developer in understanding the lifetime and scope of the launched coroutine.

4.3 Querying Asynchronous Objects

A race condition that commonly occurs with many languages is the result of querying the state of an object in shared memory and, based on that, deciding how to act on it. Listing 9 provides such an example where the developer checks if the channel is closed and sends a message if it is not closed.

Root cause. The root cause for this bug is simple and similar to race conditions in classical multithreaded programs. When an object is shared among threads, its state might change between checking its state and acting on it depending on the accesses to the object. In the example of Figure 9, the channel can be closed between the check `isClosedForSend` (line 1) and sending the message (line 2). A visual representation of this particular example is provided in Figure 3.

Misconceptions. The misconception is that the state of such an object will not change between the query and the action, disregarding the possibility of interleavings from other threads. The fix for the bug is to avoid sending if the channel is closed. As locking is undesired in Kotlin coroutines, the bug fix does not introduce explicit synchronization but wraps the `send` call in a `try/catch` block. A call on a closed channel should throw a `ClosedSendChannelException`, which allows the developer to handle it gracefully.

Automated detection. As the bug occurs in the case of a race condition on the channels, detecting this pattern can benefit from existing data race detectors.

4.4 Synchronizing with Cancellation

Synchronization with Cancellation is an attempt to ensure that a certain coroutine only runs once at a time by canceling the previous coroutine before the next is launched.

Root cause. The `cancel` method of a coroutine returns before the coroutine actually cancels or stops. In other words, `cancel` cannot be used to synchronize executions. In the example of Listing 11 there is a coroutine launched on line 7. The desired behavior is that this coroutine runs only once at a time. Otherwise, there exists a possible data race between the `read` (line 8) and `write` (line 11) actions. The `refresh` function (line 3) might be called again before the coroutine on line 7 is finished. To prevent a second coroutine from running in parallel, the developer keeps a reference to the `Job` of that coroutine and cancels it before a new

Listing 9 Race condition on channel status, taken from the bug fix in [24].

```
1 if (!channel.isClosedForSend) {
2     channel.send(message)
3 }
```

Listing 10 Potential solution to Listing 9.

```
1 try {
2     channel.send(message)
3 } catch (e: ClosedSendChannelException) {
4     // handle closed channel if needed
5 }
```

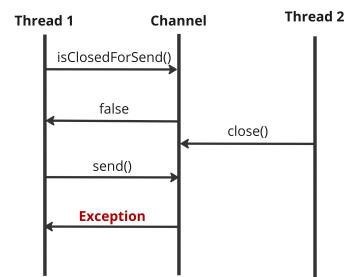


Figure 3 Visual representation of the execution order that could lead to a send operation over an unexpected closed channel.

Listing 11 Missed synchronization with `Job.cancel`, a simplified version of the bug in [5].

```
1 var pendingJob: Job? = null
2
3 suspend fun refresh() {
4     pendingJob?.cancel() // does not wait for the coroutine to stop
5     coroutineScope {
6
7         pendingJob = launch(Dispatchers.IO) {
8             val result = read(someVar)
9
10            launch(Dispatchers.Main) {
11                write(someVar, result)
12            }
13        }
14    }
15 }
```

coroutine is started. This should ensure that the coroutine only runs once at a time. However, cancellation does not guarantee that execution will stop immediately, and the `cancel()` call does return immediately. Therefore, this coroutine can exist in parallel. Primitives that allow for synchronization are mutexes, joins, and channels. In this case, a potential solution is provided in Listing 12, a mutex that wraps the scope of the launched coroutine will make sure this coroutine can only be launched once the previous one is finished.

Misconceptions. A developer might be unaware that canceling of coroutines is cooperative, meaning that they can only cancel and stop when they reach a suspension point or manually check their cancellation status. Therefore, canceling the coroutine never guarantees that it actually stops. Additionally, the `cancel` method does not wait for the coroutine to be stopped.

4.5 Swallowing CancellationException

Incorrect handling of `CancellationExceptions` can introduce bugs manifesting in the executions with exceptions. While these bugs are not strictly concurrency bugs, they are specific to Kotlin coroutines. Therefore, we cover them in this section.

 **Listing 12** A potential solution to the synchronizing with cancel example in Listing 11.

```

1 val mutex = Mutex()
2 var pendingJob: Job? = null
3
4 suspend fun refresh() {
5     pendingJob?.cancel() // optional
6     mutex.withLock { // waits for coroutine to stop
7         coroutineScope {
8
9             pendingJob = launch(Dispatchers.IO) {
10                 val result = read(someVar)
11
12                 launch(Dispatchers.Main) {
13                     write(someVar, result)
14                 }
15             }
16         }
17     }
18 }
```

Root cause. A `CancellationException` is thrown to signal that a coroutine is canceled. When this exception is caught, it interferes with the canceling mechanism of the coroutines. In Listing 13, a call to `suspendingFunction` is wrapped in a try/catch block to log any occurred errors. However, when the coroutine gets canceled while executing `suspendingFunction` a `CancellationException` is thrown which is then caught and logged. While logging a cancellation might be unfortunate, a bigger problem is that the cancellation is swallowed, since for it to work the exception needs to be propagated. A solution is given in Listing 14. The `CancellationException` is specifically caught and rethrown. We observed 14 bugs caused by accidentally swallowing `CancellationException`. This is also one of the most discussed issues in the Kotlin Coroutines issue tracker [29].

A bug that we did not observe but can occur involving `CancellationException` is when older Java frameworks throw these exceptions that could potentially run in a coroutine, making it stop silently when it shouldn't. In the example of Listing 15, a coroutine is launched on line 2. This coroutine calls a function `libraryCall` (line 3), which in this example is part of the same file but, in practice, can be any java library that throws a `CancellationException`. When this code example is executed, it will never reach the `println` statement on line 4. However, the program does finish gracefully (exit code 0). This is discussed in greater detail in the article "The Silent Killer That's Crashing Your Coroutines" [11].

 **Listing 13** Swallowed `CancellationException`.

```

1 suspend fun foo() {
2     try {
3         suspendingFunction()
4     } catch (e: Exception) {
5         Log.error(e)
6     }
7 }
```

Listing 14 A potential solution to swallowed `CancellationException` in Listing 13.

```

1 suspend fun foo() {
2     try {
3         suspendingFunction()
4     } catch (e: CancellationException) {
5         throw e
6     } catch (e: Exception) {
7         Log.error(e)
8     }
9 }
```

Listing 15 A library call throws a `CancellationException` and incorrectly cancels the coroutine.

```

1 fun main() = runBlocking {
2     launch {
3         libraryCall()
4         println("Unreachable")
5     }
6 }
7
8 fun libraryCall() { // Anything that throws CancellationException
9     throw CancellationException() // Exception unrelated to coroutines
10 }
```

Misconceptions. The developer might forget or not be aware of the canceling mechanism of coroutines. Accidentally catching a `CancellationException` is the result of that.

Possible automated detection. One may implement a static analysis that inspects the call graph and searches for `try-catch` blocks that can call a `suspend` in the `try` block and catch `CancellationException` (or a more generic one, e.g., `Exception` or `Throwable`) without propagating it by rethrowing outside the `catch` block. However, one should be careful when other constructs from the standard Java library that may throw `CancellationException` are used in the `try` block.

5 Threats to Validity

Potential threats to the validity include the representativeness of the studied concurrency bugs and our study methodology. Similar to other bug studies, our work analyzes a limited set of project repositories and a limited set of their commits.

We studied the Kotlin repositories based on our selection criteria for well-maintenance and popularity, which are based on the lines of code, number of commits, and stars. However, these criteria can potentially miss some Kotlin repositories with concurrency bugs. Similarly, we study a subset of commits in the selected repositories filtered by some keywords. We do not include some Kotlin framework keywords in our repository search, such as “channel” and “suspend”, which are frequently used during development with coroutines, appear in many of the commits, and introduce noise in the search results. Hence, the search results may potentially miss some bugs. Moreover, some bugs may not be explicitly discussed in the repositories’ commit messages or may not even have been diagnosed or fixed; therefore, they

may be missed by a repository search. Finally, our methodology involves a manual analysis of the commit source codes. While we aimed the analysis to be comprehensive, it may have missed some concurrency bugs studied or fixed in the commits.

While the study has limitations, some of which are inherent to real-world bug studies, we believe the studied bugs provide a useful sample of real-world concurrency bugs to shed light on misunderstandings and bug patterns in Kotlin programs with coroutines.

6 Key Takeaways and Discussion

While Kotlin coroutines provide a robust and straightforward mechanism for writing asynchronous programs, our study shows that developers can introduce specific concurrency bugs if they need to correctly use the asynchrony features.

Key takeaways. Our main observation is that developers may find it hard to identify *function coloring*, i.e., distinguishing which parts of their code run in asynchronous contexts, *bridging asynchronous and synchronous parts of their code*, and they may be unaware of or disregard the *semantics and mechanisms of some coroutine features* (e.g., the coroutine cancellation mechanism).

- While the Kotlin Coroutines framework helps developers identify asynchrony in their code by marking suspendable functions, programmers should be aware of regular functions that are called by asynchronous functions. Such functions, in turn, can run in an asynchronous context, and it can be hard to follow if they run in synchronous or asynchronous context, especially in large programs with deep execution call stacks.
- Developers should be careful when bridging the synchronous and asynchronous parts of their programs. Our analysis shows that when they need to call a suspend function from a synchronous context, they may tend to use *runBlocking* calls as a quick solution. This mistake is understandable since a suspend function calling a synchronous one is unaware that it might reach a *runBlocking*, while the synchronous function is unaware it is called from a coroutine. However, this unawareness can lead to dangerous *runBlocking* calls, which can result in serious concurrency errors such as deadlocks.
- When developers are aware of the asynchronous execution context but still need to call a suspend function from a normal one, they might choose to pass a coroutine scope. This solution, however, can introduce unexpected executions: the suspend function can outlive the synchronous function that called it. Incorrect reasoning about the function scopes and making incorrect assumptions about the completion of functions can result in unexpected execution orders of the program's statements.
- Similarly, the developers should be aware of the asynchronous objects and use the correct library structures to access or run operations on them. Incorrect assumptions (e.g., on the synchronization with channels) and ignorance of possible interference from other threads result in concurrency errors.
- Finally, developers should be aware of the semantics and guarantees of the programming abstractions and features they use. For example, we observed that there is common confusion about the canceling behavior of coroutines. Canceling a coroutine is cooperative and, therefore, does not guarantee it will be canceled. Similarly, the developers unaware of the cancellation exception handling mechanism of coroutines can introduce serious problems as incorrectly catching for these exceptions potentially silences critical exceptions in their programs.

Discussion. Besides increasing developers' awareness of common misconceptions, understanding common bug patterns can lead to the development of suitable program analysis tools for Kotlin programs. We communicated our findings to the Kotlin and IntelliJ teams at JetBrains. Our findings have led to the development of an inspection tool for detecting problematic `runBlocking` calls, which is currently part of the IntelliJ source code [7].

7 Related Work

7.1 Studies of Real-world Concurrency Bugs

Similar studies have been conducted that collect and categorize concurrency bugs in different programming languages and frameworks. Earlier work analyses of C/C++ concurrency bugs from server and client applications [23, 15], and report that most non-deadlock concurrency bugs are caused by atomicity and order violations. Focusing on misuse of asynchronous constructs in C# programs, the work in [26] identifies problems due to misuse or unnecessary use of asynchronous methods, invocation of long-running tasks in asynchronous methods and some anti-patterns specific to C#'s `async` and `await` model. Another study on real-world concurrency bugs [40] targets asynchronous and event-driven Node.js programs. The work identifies the concurrency bug patterns in Node.js programs as atomicity violations, order violations, and starvation in the execution of event handlers.

For Golang, the studies in [8] and [38] collect and analyze real-world concurrency bugs. The bug study in [8] focuses on data races in Go programs, and [38] focuses on the inter-thread communication mechanisms, i.e., whether message passing or shared memory concurrency is less error-prone. The findings of these works successfully led to the research and development of multiple concurrency bug analysis techniques for Go [39, 21, 13, 20, 36].

Targeting the actor model of concurrency, bug studies in [16, 22, 4] focus on actor programs, where a program consists of a set of actors that concurrently operate on their local states and communicate by exchanging asynchronous messages. The work in [16] categorizes the bugs in actor programs into communication bugs (problems in handling messages or due to delivery orderings of messages) and coordination bugs (e.g., ungraceful shutdown or recovery of actors). Following the categorization of classical shared-memory concurrency bugs, the work in [22] categorizes the actor program bugs into lack of progress and message protocol violations and defines specific subclasses of each category for actor programs. The study of actor concurrency bugs in [4] focuses on real-world Akka actor programs and analyzes their symptoms, root causes, and API usage. The bug characteristics in actor programs differ from classical shared memory programs and coroutine programs we study in this work since actors provide a high-level concurrency model without a shared state, and the concurrency nondeterminism is in the order of asynchronous events.

Different from these works, which identify the classical bug patterns of atomicity and order violations in the shared memory accesses [23, 15, 1], atomicity and order violations in the handling of events [40], message protocol violations in actor programs [16, 22, 4], or misuses and bugs in asynchronous programming in C# [26] or Go [8, 38], in this work, we focus on concurrency bugs in Kotlin programs and identify new bug patterns specifically related to Kotlin coroutines.

7.2 Analysis of Kotlin Programs

Kotlin is a relatively new programming language and only some recent research addresses the analysis of Kotlin programs. A related work targeting channel-based concurrent programs [30], which tests channel-based systems through fuzzing, has found and led to the resolution of a

bug in the Kotlin coroutine implementation. Another related work, Lincheck [19], provides a testing framework for concurrent algorithms that run on the JVM, which has been adopted in Java and Kotlin communities.

The bug patterns we discovered in this work can be useful for designing and developing program concurrency analysis tools for Kotlin programs.

8 Conclusion

This paper introduces the first real-world concurrency bug study for Kotlin coroutines, shedding light on the typical patterns of concurrency bugs. Having examined 55 concurrency bugs selected from 7 popular open-source repositories that use Kotlin coroutines, we identified common bug patterns related to Kotlin coroutine semantics. We distilled suggestions for Kotlin developers to avoid these programming errors and discussed possible techniques to detect such issues automatically. We reported our findings to the Kotlin and IntelliJ teams at JetBrains, and we believe our findings will help future research and development of concurrency analysis tools for Kotlin.

References

- 1 Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, Hans Hansson, and Wasif Afzal. 10 years of research on debugging concurrent and multicore software: a systematic mapping study. *Softw. Qual. J.*, 25(1):49–82, 2017. doi:10.1007/S11219-015-9301-7.
- 2 Automattic. Woocommerce Android app, 2023. URL: <https://github.com/shadowsocks/shadowsocks-android>.
- 3 Automattic. Wordpress for android, 2023. URL: <https://github.com/wordpress-mobile/WordPress-Android>.
- 4 Mehdi Bagherzadeh, Nicholas Fireman, Anas Shawesh, and Raffi Khatchadourian. Actor concurrency bugs: a comprehensive study on symptoms, root causes, API usages, and differences. *Proc. ACM Program. Lang.*, 4(OOPSLA):214:1–214:32, 2020. doi:10.1145/3428282.
- 5 Jeff Boek. Bugfix commit, firefox for android, November 2018. URL: <https://github.com/mozilla-mobile/firefox-android/commit/d18bfe9f1bb5f0ed0f85e5fa36cddfaee84b5d47>.
- 6 Bob Brockbernd. Found bugs per bug type, April 2024. URL: <https://github.com/bbrockbernd/kotlin-coroutine-bugs>.
- 7 Bob Brockbernd. Runblocking inspection implementation, June 2024. URL: <https://github.com/JetBrains/intellij-community/commit/ea8296d53925ec87ddbee66f37412793d3fbdb14>.
- 8 Milind Chabbi and Murali Krishna Ramanathan. A study of real-world data races in Golang. In Ranjit Jhala and Isil Dillig, editors, *PLDI ’22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 474–489. ACM, 2022. doi:10.1145/3519939.3523720.
- 9 Yi-An Chen and Yi-Ping You. Structured concurrency: A review. In *Workshop Proceedings of the 51st International Conference on Parallel Processing, ICPP Workshops 2022, Bordeaux, France, 29 August 2022 - 1 September 2022*, pages 16:1–16:8. ACM, 2022. doi:10.1145/3547276.3548519.
- 10 Sam Cooper. How I fell in Kotlin’s runblocking deadlock trap, and how you can avoid it, October 2023. URL: <https://betterprogramming.pub/how-i-fell-in-kotlins-runblocking-deadlock-trap-and-how-you-can-avoid-it-db9e7c4909f1>.
- 11 Sam Cooper. The silent killer that’s crashing your coroutines, February 2023. URL: <https://medium.com/better-programming/the-silent-killer-thats-crashing-your-coroutines-9171d1e8f79b>.

- 12 Ana Lúcia de Moura and Roberto Ierusalimschy. Revisiting coroutines. *ACM Trans. Program. Lang. Syst.*, 31(2):6:1–6:31, 2009. doi:[10.1145/1462166.1462167](https://doi.org/10.1145/1462166.1462167).
- 13 Nicolas Dilley and Julien Lange. Automated verification of go programs via bounded model checking. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*, pages 1016–1027. IEEE, 2021. doi:[10.1109/ASE51524.2021.9678571](https://doi.org/10.1109/ASE51524.2021.9678571).
- 14 Roman Elizarov, Mikhail A. Belyaev, Marat Akhin, and Ilmir Usmanov. Kotlin coroutines: design and implementation. In Wolfgang De Meuter and Elisa L. A. Baniassad, editors, *Onward! 2021: Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Virtual Event / Chicago, IL, USA, October 20-22, 2021*, pages 68–84. ACM, 2021. doi:[10.1145/3486607.3486751](https://doi.org/10.1145/3486607.3486751).
- 15 Rui Gu, Guoliang Jin, Linhai Song, Linjie Zhu, and Shan Lu. What change history tells us about thread synchronization. In Elisabetta Di Nitto, Mark Harman, and Patrick Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 426–438. ACM, 2015. doi:[10.1145/2786805.2786815](https://doi.org/10.1145/2786805.2786815).
- 16 Brandon Hedden and Xinghui Zhao. A comprehensive study on bugs in actor systems. In *Proceedings of the 47th International Conference on Parallel Processing, ICPP 2018, Eugene, OR, USA, August 13-16, 2018*, pages 56:1–56:9. ACM, 2018. doi:[10.1145/3225058.3225139](https://doi.org/10.1145/3225058.3225139).
- 17 JetBrains. IntelliJ idea community edition, 2023. URL: <https://github.com/JetBrains/intellij-community>.
- 18 JetBrains. Ktor, 2023. URL: <https://github.com/ktorio/ktor>.
- 19 Nikita Koval, Alexander Fedorov, Maria Sokolova, Dmitry Tsitelov, and Dan Alistarh. Lincheck: A practical framework for testing concurrent data structures on JVM. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part I*, volume 13964 of *Lecture Notes in Computer Science*, pages 156–169. Springer, 2023. doi:[10.1007/978-3-031-37706-8_8](https://doi.org/10.1007/978-3-031-37706-8_8).
- 20 Ziheng Liu, Shihao Xia, Yu Liang, Linhai Song, and Hong Hu. Who goes first? detecting go concurrency bugs via message reordering. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 888–902. ACM, 2022. doi:[10.1145/3503222.3507753](https://doi.org/10.1145/3503222.3507753).
- 21 Ziheng Liu, Shuofei Zhu, Boqin Qin, Hao Chen, and Linhai Song. Automatically detecting and fixing concurrency bugs in go software systems. In Tim Sherwood, Emery D. Berger, and Christos Kozyrakis, editors, *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, pages 616–629. ACM, 2021. doi:[10.1145/3445814.3446756](https://doi.org/10.1145/3445814.3446756).
- 22 Carmen Torres Lopez, Stefan Marr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. A study of concurrency bugs and advanced development support for actor-based programs. In Alessandro Ricci and Philipp Haller, editors, *Programming with Actors - State-of-the-Art and Research Perspectives*, volume 10789 of *Lecture Notes in Computer Science*, pages 155–185. Springer, 2018. doi:[10.1007/978-3-030-00302-9_6](https://doi.org/10.1007/978-3-030-00302-9_6).
- 23 Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In Susan J. Eggers and James R. Larus, editors, *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*, pages 329–339. ACM, 2008. doi:[10.1145/1346281.1346323](https://doi.org/10.1145/1346281.1346323).
- 24 Sergey Mashkov. Bugfix commit, ktor, October 2020. URL: <https://github.com/ktorio/ktor/commit/7dfa6d9f1650430738e76cba165eb3529687be3c>.
- 25 Mozilla. Firefox for android, 2023. URL: <https://github.com/mozilla-mobile/firefox-android>.

8:20 Understanding Concurrency Bugs in Real-World Programs with Kotlin Coroutines

- 26 Semih Okur, David L. Hartveld, Danny Dig, and Arie van Deursen. A study and toolkit for asynchronous programming in c#. In Pankaj Jalote, Lionel C. Briand, and André van der Hoek, editors, *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 1117–1127. ACM, 2014. doi:[10.1145/2568225.2568309](https://doi.org/10.1145/2568225.2568309).
- 27 Daniil Ovchinnikov. Runblocking should let go of CPU token before parking the thread, December 2023. URL: <https://github.com/Kotlinx/kotlinx.coroutines/issues/3983>.
- 28 Ondrej Ruttkay. Bugfix commit, woocommerce android app, January 2021. URL: <https://github.com/woocommerce/woocommerce-android/commit/6156420791b1e78067cd5dacbe5a15d0cc24979d>.
- 29 Anton Spaans. Provide a runcatching that does not handle a cancellationexception but re-throws it instead, February 2020. URL: <https://github.com/Kotlin/kotlinx.coroutines/issues/1814>.
- 30 Quentin Stiévenart and Magnus Madsen. Fuzzing channel-based concurrency runtimes using types and effects. *Proc. ACM Program. Lang.*, 4(OOPSLA):186:1–186:27, 2020. doi:[10.1145/3428254](https://doi.org/10.1145/3428254).
- 31 Mygod Studio. Shadowsocks for android, 2023. URL: <https://github.com/shadowsocks/shadowsocks-android>.
- 32 Petr Surkov. Bugfix commit, woocommerce android app, March 2024. URL: <https://github.com/woocommerce/woocommerce-android/commit/5dbb3b1834f67f097be7db96cab04c3ca99d7294>.
- 33 Don Syme, Tomas Petricek, and Dmitry Lomov. The f# asynchronous programming model. In Ricardo Rocha and John Launchbury, editors, *Practical Aspects of Declarative Languages - 13th International Symposium, PADL 2011, Austin, TX, USA, January 24-25, 2011. Proceedings*, volume 6539 of *Lecture Notes in Computer Science*, pages 175–189. Springer, 2011. doi:[10.1007/978-3-642-18378-2_15](https://doi.org/10.1007/978-3-642-18378-2_15).
- 34 Martin Sústrik. Structured concurrency, February 2016. URL: <https://250bpm.com/blog/71/>.
- 35 Tachiyomiorg. Tachiyomi, 2023. URL: <https://github.com/tachiyomiorg/tachiyomi>.
- 36 Saeed Taheri and Ganesh Gopalakrishnan. Automated dynamic concurrency analysis for go, 2021. arXiv:[2105.11064](https://arxiv.org/abs/2105.11064).
- 37 Andrew Troelsen and Andy Olsen. *Pro C# 5.0 and the .NET 4.5 Framework*, volume 6. Springer, 2012.
- 38 Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiyang Zhang. Understanding real-world concurrency bugs in go. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 865–878. ACM, 2019. doi:[10.1145/3297858.3304069](https://doi.org/10.1145/3297858.3304069).
- 39 Oskar Haarklou Veileborg, Georgian-Vlad Saioc, and Anders Møller. Detecting blocking errors in Go programs using localized abstract interpretation. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, pages 32:1–32:12. ACM, 2022. doi:[10.1145/3551349.3561154](https://doi.org/10.1145/3551349.3561154).
- 40 Jie Wang, Wensheng Dou, Yu Gao, Chushu Gao, Feng Qin, Kang Yin, and Jun Wei. A comprehensive study on real world concurrency bugs in node.js. In Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen, editors, *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 520–531. IEEE Computer Society, 2017. doi:[10.1109/ASE.2017.8115663](https://doi.org/10.1109/ASE.2017.8115663).

A Language-Based Version Control System for Python

Luís Carvalho  

NOVA LINCS, NOVA School of Science and Technology, Caparica, Portugal

João Costa Seco  

NOVA LINCS, NOVA School of Science and Technology, Caparica, Portugal

Abstract

We extend prior work on a language-based approach to versioned software development to support versioned programs with mutable state and evolving method interfaces. Unlike the traditional approach of mainstream version control systems, where a textual diff represents each evolution step, we treat versions as programming elements. Each evolution step, merge operation, and version relationship is represented explicitly in a multifaceted code representation. This provides static guarantees for safe code reuse from previous versions and forward and backwards compatibility between versions, allowing clients to use newly introduced code without needing to refactor their program manually. By lifting versioning to the language level, we pave the way for tools that interact with software repositories to have more insight into a system’s behavior evolution. We instantiate our work in the Python programming language and demonstrate its applicability regarding common evolution and refactoring patterns found in different versions of popular Python packages.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Program semantics

Keywords and phrases Software evolution, type theory

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.9

Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact):*

<https://doi.org/10.4230/DARTS.10.2.3>

Funding This work is supported by EU Horizon Europe under Grant, Agreement no. 101093006 (TaRDIS), NOVA LINCS UIDB/04516/2020 (<https://doi.org/10.54499/UIDB/04516/2020>) and UIDP/04516/2020 (<https://doi.org/10.54499/UIDP/04516/2020>) with financial support of FCT.I.P.

1 Introduction

The evolution of software systems is an essential aspect of software development and its life-cycle. As requirements from stakeholders change, software systems must evolve to conform to such changes. These changes may include bug fixing, implementing new features, porting code to new hardware, updating business requirements, and other tasks that represent activities in the life-cycle of a software system. As the release cycles in the software development process become shorter [20] the overhead of managing multiple versions increases. On the one hand, software maintainers have to reason more frequently about what changes to backport and whether the changes they introduced break existing client code. On the other hand, the stakeholders of a product will need to consider more frequently whether or not to upgrade. Integrating a release with breaking changes leads to runtime errors and requires manual intervention, while missing a critical update may lead to software vulnerabilities.

Given the manual effort required for semantic software versioning, which is largely rooted in the fact that version control systems (VCS, e.g. `git`, `svn`, `mercurial`) operate on *text* rather than *programs*, we extend prior work on a language-based VCS for a core functional language, Versioned Featherweight Java [5], by adding support for programs with mutable state and side effects, and versioning of class methods.

 © Luís Carvalho and João Costa Seco;
licensed under Creative Commons License CC-BY 4.0
38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 9; pp. 9:1–9:27



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

9:2 A Language-Based Version Control System for Python

The current industry practices for software evolution advocate the use of version control systems. However, while very good at managing *changes* to information, VCS give no semantic meaning to each program delta (*diff*). Each evolution step is typically defined by 1) the new code it introduces and 2) an accompanying natural language message describing the change. To issue a new version of the software, the developer includes one or more of these steps and generates a *changelog*, naming the version according to some convention.

A widely adopted convention is Semantic Versioning [26], where the increment of each version identifier denotes the nature of the introduced changes. Clients can then define their update policy according to this convention. There is no guarantee that the code effectively follows the versioning policy (e.g. the developer unknowingly introduced an unexpected breaking change), thus VCS allow for inconsistent versions to be committed, and still require work from developers in identifying the kind of changes made [33, 27].

In this work, we embed the versioning in the programming language, so that the developer specifies as code what each delta is, and also how it relates to other versions. This allows for (formal) verification on if and how the different versions interact with each other; it allows for clients to use newly introduced code without changing their existing program; it is well-suited for targeting software for a specific version, producing artifacts containing only the necessary code; for providing a version-aware development environment (drawing inspiration from [23]) in which a developer may edit a snapshot and have the changes automatically committed with the appropriate version tags. We extend previous work [5] to provide a language-based VCS for Python programs, where versions and their relations are specified as code in the program. The features that strictly extend [5] are:

- A mechanism for defining transformations (which we call *lenses*) between method interfaces in different, related, versions. This allows the developer to specify how the evolution of a method interface is to be handled by clients, so that they do not have to manually adapt their code to account for the new definition.
- Support for mutability and side-effects. Featherweight Java (FJ) is a functional language and, as such, does not model side effects. In this work, we instantiate our core calculus in Python, a mainstream language with new challenges in comparison to FJ, particularly concerning mutability. To do so, we ensure that mutability and side-effects are well reflected when transitioning between versions with different state representations.

In this setting, the developer of a versioned program defines how the versions of the program relate to each other, and provides each evolution step as code, to define how clients should evolve between versions. As such, clients can then get new features from other versions without their code breaking, and without any need for manual refactoring: the evolution steps provided by the developer are used to adapt the code to the client version, so that they can use it without breaking.

The diagram in Figure 1 provides the intuition on the parallels between traditional version control systems and our approach. With the developer providing each evolution step as code, we allow clients to migrate automatically, thus removing the need for migration tools (which can also introduce bugs) to help clients do that. The traditional repository operations, such as committing a file, or merging two branches, are well supported in our setting, by defining the appropriate version graph. Finally, we provide a slicing procedure to allow developers to issue a release for a given version, without having to manually perform operations on the repository (e.g. backporting a security fix to another branch). The resulting slice, corresponding to a release for a given version, ensures that the release conforms to the versioning graph, i.e. it does not introduce unintended breaking changes.

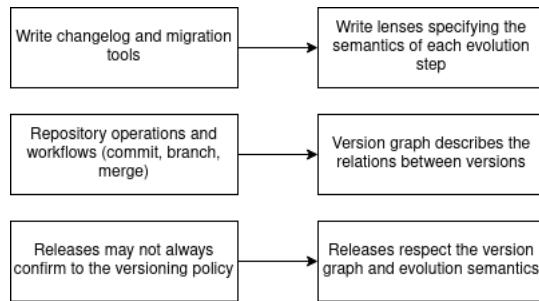


Figure 1 Diagram describing the parallels between version control systems and our approach.

The main contributions of this paper are as follows:

- Extending the work in [5], with method lenses and programs with state and side-effects.
- Instantiating the core calculus in Python to add support for versioned programs.
- Extending a type system for Python to account for versioned programs.
- Introducing a slicing compiler that can generate a projection of a versioned Python program for a single target version.

We expect this approach to provide static guarantees of safe code reuse from previous snapshots, as well as ensuring forward and backward compatibility between related versions through type-safe state transformation functions.

The remainder of this document is structured as follows: section 2 provides a running example to illustrate the ideas in this work; section 3 describes in detail our technical approach; section 4 evaluates the approach using public Python packages and provides the empirical results; section 5 discusses the related work in this space; section 6 discusses the limitations of this work and how we plan to address them; section 7 summarises our results.

2 Example

In Figure 2, we present examples of two Python programs: a versioned library program (Figure 2a) and a client program that uses a specific version of that library (Figure 2b). The versioned program (Figure 2a) contains a version graph describing how the different versions relate to each other (lines 1-3). The class `Name` of this program contains versioned definitions of methods, which are annotated with the version in which they are introduced.

In this example, there are three different versions of the library program, the class `Name`. The program starts with the definition of a version graph for class `Name`. Version `init` is the starting point of the example, and includes the definitions of fields `first` and `last`, and of methods `display` and `set_last`. Note that, in version `bugfix`, the developer introduces a new definition of method `display` (lines 15-17), but otherwise makes no changes. This new definition supersedes the previous one for clients running in the context of version `init`. This is indicated by the `replaces` relationship between versions `bugfix` and `init`.

Finally, in version `full` the developer introduces a new constructor (lines 9-11), where they change the internal state representation of the class (line 11), and a new method, `get_full_name` (lines 21-23). Contrary to version `bugfix`, these new definitions are only meant for clients that are specifically running in the context of version `full`, and does not affect clients in other versions. The code from version `init` is available in the new versions, allowing it to be safely reused.

9:4 A Language-Based Version Control System for Python

```

1  @version('init')
2  @version('bugfix', replaces=['init'])
3  @version('full', upgrades=['init'])
4  class Name:
5      @at('init')
6      def __init__(self, first: str, last: str):
7          self.first = first
8          self.last = last
9      @at('full')
10     def __init__(self, full: str):
11         self.fname = full
12     @at('init')
13     def display(self):
14         return f'{self.first}, {self.last}'
15     @at('bugfix')
16     def display(self):
17         return f'{self.last}, {self.first}'
18     @at('init')
19     def set_last(self, name: str):
20         self.last = name
21     @at('full')
22     def get_full_name(self):
23         return self.fname

```

(b) Client code for version full.

```

1  @run('full')
2  def main():
3      n = Name('Bob Dylan')
4      n.set_last('Marley')
5      print(n.display())
6      print(n.fname)

```

(c) Lenses for fields between different versions.

```

1  @get('full', 'init', 'first')
2  def lens_first(self) -> str:
3      if ',' in self.fname:
4          return self.fname.split(',') [0]
5      return self.fname
6  @get('full', 'init', 'last')
7  def lens_last(self) -> str:
8      if ',' in self.fname:
9          return self.fname.split(',') [1]
10     return ''
11  @get('init', 'full', 'fname')
12  def lens_full(self) -> str:
13      return f'{self.first} {self.last}'

```

(a) Example of a versioned Python class.

(c) Lenses for fields between different versions.

 **Figure 2** Client and library code.

The version graph (lines 1-3) is a set of version decorators defining a name for a new version and the type of relationship (`upgrades` or `replaces`) they have with other versions. The type of relationship describes how the new version affects the existing version graph, including versions introduced earlier: if the code is to be implicitly available for clients in previous (related) versions, then the type `replaces` is used. This ensures clients running in a previous version will have the new definitions available without having to update their code (e.g. bug or security fixes). Otherwise, if the new version is a backward-incompatible extension of a previous one (c.f. class inheritance), then the type `upgrades` is used. This ensures that definitions introduced in an `upgrade` version are only available for clients running explicitly in that version. These decorators can be provided by the developer (i.e. while developing the program) or, for existing programs, they can be (naively) inferred from the repository (e.g. the version graph for the repository in Figure 3a is depicted in Figure 3b).

The library program also contains several versioned programming elements, such as constructors (lines 5-11) and methods (lines 15-23), which are decorated with the version in which they are introduced.

The client program (Figure 2b) uses version `full` of the library. It uses the `__init__` method (line 3) introduced in version `full`; the `set_last` (line 4) and `display` (line 5) methods introduced in previous versions; and the attribute `fname`, which is the only field of class `Name` in version `full`.

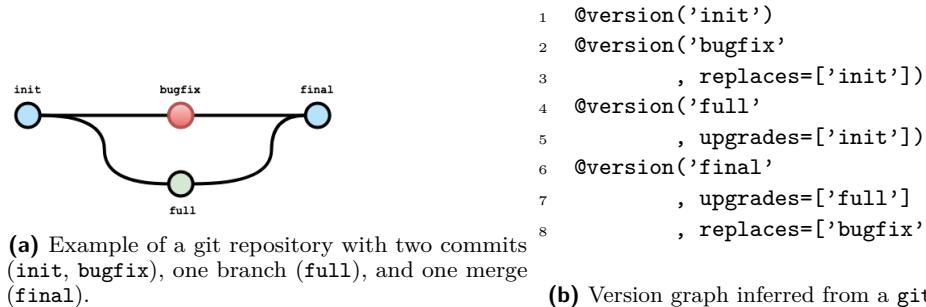


Figure 3 Repository and its corresponding version graph.

To resolve methods for the client at version `full`, we perform a method lookup operation taking the version graph into account: the available definition of method `set_last` is the one introduced in version `init`; the definition of method `display` is the one from version `bugfix`, because it replaces the definition in version `init`. Given that these definitions are available at version `full`, clients should be able to safely use them in that context.

However, since these methods use a different internal representation of class `Name`, we can not use them as-is in version `full`, as that would result in an error since the class fields do not match. We propose that, instead of re-implementing all the methods missing from the new representation (of version `full`), the developer provides a mapping between the fields of versions `init` and `full`. This mapping, which we call a *get lens* (Figure 2c), represents the inner semantics of the evolution step.

A *get lens* is introduced by a method with a decorator of the form `@get(v, v', f)`, that maps how field `f` of version `v'` is derived from the state in version `v`. For instance, the lens from version `full` to version `init` of field `first` (Figure 2c, lines 4-8) defines how the field `first`, in version `init`, is obtained from the state of version `full`. Lenses are the building blocks for developers to express evolution steps in the form of code.

The versioning-aware method lookup policy coupled with lenses allows clients to use code from different versions at runtime in a way that respects the version graph and the evolution semantics specified by the developer.

It is also possible to obtain a static, self-contained, slice of a versioned program for a specific target version. This slice will include all code available throughout the version graph for that target version, using the lookup policy described earlier. Again, since this can include code that uses different internal state representations, the slicing procedure rewrites such expressions using the corresponding lenses so that they are correct in the context of the target version, as hinted at earlier.

For example, at version `init`, the available definition for method `display` is the one introduced in the `bugfix` version. As such, this definition must be included in the slice for version `init` (Listing 1). Since both versions `init` and `bugfix` share the same internal representation of class `Name` (version `bugfix` does not define its own class fields), in the slice for version `init` we do not need to rewrite the method `display`, as it already complies with the state of version `init`.

```

1 class Name:
2     def __init__(self, first: str, last: str):
3         self.first = first
4         self.last = last
5     def display(self):
6         return f'{self.last}, {self.first}'

```

Listing 1 Slice of a versioned Python class for version `init`.

At version `full`, the available definition for methods `display` and `set_last` are introduced in versions `bugfix` and `init`, respectively. These definitions must be included in the slice for version `full` (Listing 2). In this case, since version `full` introduces a new internal representation of class `Name` (by changing the fields from version `init`), then the definitions of methods `display` and `set_last` do not conform to this representation, as they are defined in the context of other versions. As such, we need to rewrite the methods using the corresponding lenses for fields `first` and `last` (Listing 2, lines 12-21), so that they match the context of version `full`.

```

1  class Name:
2      def __init__(self, full: str):
3          self.fname = full
4      def display(self):
5          return f'{self.lens_last()}, {self.lens_first()}'
6      def set_last(self, name: str):
7          __name = name
8          self.fname =
9              self.lens_full(first=self.lens_first(), last=__name)
10     def get_full_name(self):
11         return self.fname
12     def lens_full(self, first, last):
13         return f'{first} {last}'
14     def lens_first(self):
15         if ' ' in self.fname:
16             return self.fname.split(' ')[0]
17         return self.fname
18     def lens_last(self):
19         if ' ' in self.fname:
20             return self.fname.split(' ')[1]
21         return ''
```

Listing 2 Slice of a versioned Python class for version `full`.

Both the library and client programs are valid Python programs¹. The library program can be fed as input to our compiler to extract a projection for a given version. The result of this is a valid Python program without any version annotations. The client program, when fed to an interpreter, is executed following the operational semantics described in this work for multi-version program execution.

3 Design

The ideas presented in this work are mainly language-agnostic, provided the language has support for objects and mutability. To implement these ideas, we instantiate this work in the Python programming language [1]. We chose Python for the following reasons:

- Being a mainstream and widely adopted language facilitates the evaluation of the approach using publicly available repositories of both software libraries and their corresponding clients.

¹ For brevity, the statement to import the decorators, which is necessary for the code to run, is omitted here

- In comparison with other mainstream languages that provide similar features (e.g. Java, C#), Python’s less complex syntax usually results in simpler programs [22]. This, again, facilitates the empirical evaluation of the approach, as the code patterns will be simpler to grok.
- From an implementation point of view, Python’s dynamic nature allows us to quickly prototype a solution that implements the ideas discussed in this work.

Our implementation works with a large subset of the Python language. Particularly, we do not provide semantics for the versioning of: `async/await` statements, `yield` statements, list comprehensions, and modules². The implementation also works under the assumption that the program is typed, either manually or with the help of automated type inference tools [6, 17, 32, 21]. This requirement should not be deemed too restrictive, since the practice of providing type annotations in Python programs is becoming increasingly common [7].

On top of this, we provide a type-checker and a slicing procedure for versioned Python programs. The type-checker is an extension of `Pyanalyze`, a type-checker developed by Quora that also annotates the AST nodes with their corresponding types. We extend `Pyanalyze` with support for versioned lookup of fields and methods, ensuring the correct type are inferred. The type-checker ensures the soundness of the program against its version graph, and provides the following guarantees for well-typed programs:

- A client that follows the versioning policy, defined in the version graph, will never have their code break.
- If a method needs to be rewritten for a different, related, version, it will always succeed and never produce a type error.

The slicing procedure allows for the projection of code to a specific version, inspired by software product lines and other technical approaches, such as programming variability and CI/CD pipelines. This procedure is implemented on top of a rewriting mechanism to allow library developers to release the code targeting a specific version. For well-typed programs, the slicing procedure ensures that:

- All necessary code for the target version, according to the specification in the version graph, is included.
- All the code included from other, related versions, is well-typed in the context of the target versions, by applying state or method transformations when necessary (using lenses).
- Client code that targets the sliced projection will always type-check, even if the resulting slice includes code from other versions.

The remainder of this section is structured as follows:

- Section 3.1 describes how to define versioned elements in a program.
- Sections 3.2 and 3.3 describe the versioned lookup disciplines for fields and methods respectively.
- Section 3.4 describes the use of lenses, particularly how they affect the result of a slice for a version to ensure that it is well-typed in the presence of elements from different, related versions.
- Section 3.5 describes the rewriting procedure, which is crucial to ensure that code from different versions included in a slice is always well-typed.
- Section 3.6 describes the details of the slicing procedure, that allows library developers to produce the code that targets a specific version.

² Although we do not provide versioning semantics for these elements, they can still be used in a versioned program; however, they will not follow the semantics described here.

3.1 Versioned programming elements in Python

To add support for versioning elements in Python programs, we provide the following class and function decorators³. The motivation for using decorators is that they are enough to implement the semantics described here, without requiring changes to the language syntax:

```

1  @version(<version>, <replaces>, <upgrades>)
2  @at(<version>)
3  @get(<from>, <at>, <name>)
4  @run(<version>)

```

 **Listing 3** Decorators for versioned Python programs.

These decorators allow programmers to define new versions of a class (line 1) by providing a name (`<version>`) and the relation to other versions, if any (`<replaces>` and `<upgrades>`); to introduce class methods in a version (line 2); to define class lenses that map how a field or a method (`<name>`), defined in the context of a version (`<at>`), is mapped to the context of another version (`<from>`) (line 3); and to indicate that a function should run in the context of some version (`<version>`) (line 4).

The type-checker ensures the soundness of the decorators presented here. It ensures that all version references are defined; that the version graph is acyclic; that the attribute specified in a lens (`<name>`) exists in the context of its `<at>` version, if it corresponds to a field name, and that it exists on both versions, if it corresponds to a method name; that the return type of a `get` lens matches the type of its corresponding field, or the type of its corresponding method (`<name>`); and that a class method defined at some version (`<version>`) is type-checked against the context of that same version (namely, when resolving field and method types).

The `@run` decorator defines the operational semantics to provide an environment for multi-version program execution. Additionally, we provide static semantics for versioned program slicing, so that developers are able to extract a static projection of code for a specific version. This is based on the concepts of class field and method lookup, version lenses (provided by the developer), and program slicing. We present these concepts in greater detail in the following sections.

3.2 Class field lookup

When type-checking a method of a class defined at some version v , we might encounter field access expressions (e.g. `return self.f`). To check such expressions, we need to know 1) if the field exists in that version of the class and 2) what its type is. Since fields can be defined across multiple versions of the same class, the standard (syntactic) field lookup discipline will not yield the correct field set for a given version.

As such, to be able to type-check a program, we need to define a lookup policy for class fields at a given version ($fields(C, v)$), so that we know which fields are or are not available at that version, and what their types are.

In our setting, a version (v) of a class (C) may either redefine its fields (e.g. by introducing or removing a field) or inherit the fields of related versions. The version(s) in which the fields of v are defined are called the *base versions* of v . Each version in the graph has one or more corresponding base versions for a given class ($bases(C, v)$).

³ For readers unfamiliar with Python, decorators are a method of applying a transformation to a function or a class. Except for the `run` decorator, none changes the decorated function or class (i.e. they act as syntactic hints to infer the version context.)

A field is defined in a version of a class by assigning, in any method at that version, a value to an attribute of the method caller⁴ (this is the first parameter of the method, typically named `self`).

This is expressed in rule (FIELDS-AT) (Figure 4). We start by collecting methods available at version v ($methods(C, v)$) and selecting only those tagged for version v ($at(m) = v$, where at performs a syntactic lookup of the method version decorator). Then, we collect all parameters to this method ($parameters(m)$), and inspect the method's body to check if there is an assignment to an attribute of the first parameter ($A_0.f = e \in body(m)$), which corresponds the class instance, and, if so, we collect its type (T). Finally, we check all previous related versions (W) to ensure that, if f is defined in a previous version, its type is different than the type defined at v ($C \vdash_w f : T' \wedge T \neq T'$). If the type is the same, the field is considered to be inherited, and not explicitly defined at v .

Note that, in this and all subsequent inference rules, we use some helper functions (at , $args$, $parameters$, $upgrades$, $replaces$, and $body$) that perform standard syntactic lookup of nodes in the AST (e.g. $parameters$ returns all parameters for a given method definition).

For instance, in Figure 2a (lines 7 and 8), fields `first` and `last` are introduced in version `init`, in the constructor of that version. Note that the field is only considered to be introduced in this version (as opposed to inherited) if it is not a field, with the same type, of any parent version. We make a small exception (not expressed in rule (FIELDS-AT) for brevity) for the constructor: there, a developer can redefine fields with the same name and type from other related versions⁵.

To lookup the bases of a version v , we start by collecting the fields defined explicitly at that version given the procedure described earlier ($fields_at(C, v)$). If this set is not empty, then the base of version v is simply itself (rule (BASE-SELF)). Otherwise ((rule (BASES))), the bases of v are the union of bases from the versions it upgrades and replaces (W), using the same lookup logic.

Finally, to lookup the (versioned) fields for a class C in version v ($fields(C, v)$), we collect the union of fields in all base versions of v (rules (FIELDS), (FIELDS-SELF)).

Note that, in this setting, these lookup rules are different from those presented in [5]. In particular, in this work, a version of a class can have *multiple* base versions, as opposed to a single one: as such, the lookup logic for fields is also slightly different, since we can lookup fields on multiple base versions.

In the example of Figure 2a, the base version of `bugfix`, in which no fields are explicitly defined, is `init`; the base version of `full` is itself. If we were to add a new version to this program, `final`, that merges versions `full` and `bugfix`

```
@version('final', upgrades=['full', 'bugfix'])
```

then the base versions of `final` would be versions `full` (the base of itself) and `init` (the base of `bugfix`); its fields would be the union of all fields in these versions (`first`, `last`, and `fname`).

The base versions are used in all typing and reduction rules to ensure that the version graph is respected when resolving class fields..

⁴ This follows the approach of most Python type-checkers, such as MyPy.

⁵ This is to account for the *semantic* (as opposed to syntactic) evolution of a field, when its name and type are still preserved.

9:10 A Language-Based Version Control System for Python

$$\begin{array}{c}
 m \in methods(C, v) \quad at(m) = v \quad A = parameters(m) \\
 A_0.f = e \in body(m) \quad \Gamma \vdash_v e : T \quad W = upgrades(v) \cup replaces(v) \\
 \frac{\forall_{w \in W} : f \in fields_at(C, w) \Rightarrow C \vdash_w f : T' \wedge T \neq T'}{f \in fields_at(C, v)} \text{ (FIELDS-AT)} \\
 \\
 \frac{\#fields_at(C, v) \neq 0}{v \in bases(C, v)} \text{ (BASE-SELF)} \quad \frac{W = upgrades(v) \cup replaces(v)}{\forall_{w \in W} : bases(C, w) \subset bases(C, v)} \text{ (BASES)} \\
 \\
 \frac{bases(C, v) \neq \{ v \} \quad W = bases(C, v)}{\forall_{w \in W} : fields(C, w) \subset fields(C, v)} \text{ (FIELDS)} \\
 \\
 \frac{bases(C, v) = \{ v \}}{fields(C, v) = fields_at(C, v)} \text{ (FIELDS-SELF)}
 \end{array}$$

Figure 4 Inference rules for field and base version lookup.

3.3 Method lookup

Similar to class fields, class methods can be defined across multiple versions of the same class. As such, the standard method lookup discipline will not yield the correct definition of a method for a given version, since there can be multiple definitions with the same name across different versions. Consider the following (abstract) example of a class with three related versions. Version 1 introduces a definition for methods `n` and `m`. The other versions introduce a definition for method `m`:

```

@version('1')
@version('2', upgrades=['1'])
@version('2.1', replaces=['2'])
class C:
    @at('1')
    def n(self): ...
    @at('1')
    def m(self, x): ...
    @at('2')
    def m(self, x): ...
    @at('2.1')
    def m(self, y): ...

```

Listing 4 Evolution of a method between versions.

The intuition here is that clients running at version 1 should use the definition of `m` introduced in that version, since there is no definition of `m` that replaces the one from version 1. Clients at version 2 should use the definition of `m` introduced at version 2.1, since this is declared as a replacement over version 2; and clients at version 2.1 should use the definition of `m` introduced in that version. For method `n`, all versions use the definition from version 1, which is local to version 1 and inherited in version 2 (and, subsequently, in version 2.1).

To comply with the version graph, we must also define a lookup policy for class methods at a given version, as illustrated above. The reader may notice that the definition of `m` introduced in version 2.1 – which should be available to clients in version 2 – has a different interface from the local definition of `m` at version 2 (parameter `x` is renamed to `y`). Intuitively, this means that client code is written for version 2, which may call method `m` using keyword

arguments (i.e. `C().m(x=...)`), which type-checks against the local interface, cannot simply use the new definition, as that would introduce a type error (no parameter named `x`, missing parameter `y`).

As such, the lookup of a method `m` at version `v` must return both the interface (to comply with clients targeting `v`) and its implementation (to comply with new definitions introduced in replacement versions). The lookup discipline for methods works in the following order:

Local definition. Search for a local definition of `m` introduced at version `v`. If any is found, that definition corresponds to the interface and implementation of `m` for version `v`.

Parent definition. If no local definition was found, search all versions that `v` either upgrades or replaces for a definition of `m`. If any is found, that definition corresponds to the interface and implementation of `m` for version `v`. If there are multiple matches, they must be the same definition. Otherwise, a conflict occurs, and the program is not well-typed.

Replacement definition. Finally, search all replacement versions of `v` for an implementation of `m`. If no interface was found yet (either locally or inherited from a parent version), it means `m` was introduced in a replacement version, so that interface is the one available to clients at version `v`. In any case, if there are multiple matches, they must be the same definition. Otherwise, a conflict occurs and the program is not well-typed.

This lookup policy is illustrated in the example above, where, for version 2, the interface of `m` is the one from the local definition at 2, and the implementation is from the definition at version 2.1. Later in this section, we describe how to use one interface with a different implementation. For now, it's important to retain that the lookup of methods must respect the version graph, and take into account how the client code is typed (i.e., against which interface). For method `n`, the interface and implementation at version 2 is the one inherited from version 1.

This lookup policy is used to type-check function calls at a given version; to detect missing lenses between methods of different versions, when the interface differs from the interface of the implementation (e.g. method `m` at version 2 in the previous example); to detect conflicts in the version graph; to select method definitions when providing a slice for a target version; and to find the code to execute at runtime.

3.4 Version lenses

The lookup policy for class methods, described earlier, allows for a version `v` of a class to use methods introduced in another (parent or replacement) version `t`. In such cases, there are two situations where we need to pay special attention:

- The implementation provided by the lookup policy is defined at another version, `t`, which has a different state representation (i.e. different base versions from `v`).
- The interface provided by the lookup policy is different than the interface of the implementation, and the signatures of the interfaces differ (e.g. method `m` in version 2 of the previous example).

In the first case, since the state representation of the class is different, the method body may not comply with the representation of version `v`: this is illustrated in Figure 2a, where method `display`, available for version `full`, complies with the state of another version (`init`). Intuitively, this means that, to use such implementations, we must introduce a mapping between the fields used in the method body and the fields of the target version (in this case, version `full`), otherwise we would introduce type errors.

In the second case, since the interface of the method is different from the interface available for clients at version `v`, we can not simply use the new definition, since client code is typed against a different interface (as such, doing so would introduce a type error). Intuitively, this means that, to use the new implementation, while preserving the type correctness of clients at `v`, we must introduce a mapping between the two interfaces.

These mappings, called *lenses*, for fields and methods, are described in more detail in the remainder of this subsection.

3.4.1 Field lenses

Consider again the example in Figure 2a, where the interface and implementation of method `display` for version `full` is from version `bugfix` (Figure 2a, line 17). In version `full`, class `Name` has no `first` nor `last` field, so this definition of method `display` is not well-typed in version `full`. For a client in version `full` to correctly use this code, we need to rewrite the method body, so that it complies with the desired state.

The type system requires that the developer defines the necessary lenses at version `full` for the fields at version `init` (Figure 2c). Intuitively, these lenses express how each field evolved from the state of version `full`. In this case, the lenses are simple: split the full name on a whitespace, if any, and return the corresponding component (if it exists). Later, when projecting the code for version `full`, the lenses are used to rewrite field expressions in the method body so that they are well-typed in the context of version `full`.

Each field lens is a standard class method, annotated with a `@get` decorator, of the form `@get(<at>, <from>, <name>)` (Listing 3, line 3), with a single argument (the method caller, `self`). In practical terms, the implementation of the lens answers the question: “In the context of version `<at>`, how do I represent the field `<name>` of version `<from>?`”. The type system ensures that the body of a lens is type-checked in the context of its `<at>` version (in this case, version `full`); that `<name>` is a field in version `<at>`; and that the return type of the lens matches the type of field `<name>` in version `<at>` (in this case, `str`).

Field lenses are a suitable mechanism for modelling common software evolution patterns, such as renaming a field, changing its type, or refactoring its representation (as in the example of Figure 2a, where we join both fields in a single one).

Evolution patterns that mostly concern text manipulation (as opposed to program semantics), are typically considered breaking changes (e.g. changing the name of a field in a class). In our setting, these patterns are well supported, as it is always possible for the developer to express such a pattern in the form of a lens. If they do so, then these patterns can be applied successfully without introducing breaking changes.

For instance, a lens to rename a field (`f`, to `t`) from version 1 to 2 is expressed by:

```

1  @get('1', '2', 't')
2  def rename_f_t(self):
3      return self.f
4  @get('2', '1', 'f')
5  def rename_t_f(self):
6      return self.t

```

and allows clients in version 1 to use code from version 2, thus making it a non-breaking change; and for code in version 2 to reuse code from version 1, so that the developer does not need to rewrite methods that use field `f`. In these cases, the lenses can be synthesised with the help of editor tools, instead of manually implemented by developers (c.f. refactoring tools in Bides)⁶.

⁶ This is discussed in greater detail in section 6.

Evolution patterns that concern program semantics, such as changing the type of a field, are not always possible to model with a lens. For instance, in the previous example (Figure 2c), we showed how to model such a pattern (refactoring two fields into one).

However, consider the case where we want to refactor a list (in version 1) into a dictionary (in version 2, that replaces 1). Assuming the semantics of the program dictate that the elements of the list correspond to the values in the dictionary, then we can devise a lens that maps the dictionary to the list (e.g. `return list(self.data.values())`, where `data` is the dictionary), which allows the developer to reuse code from version 1 while working in the context of version 2. But what about the other way around? If we want to map the list to a dictionary, it may not be possible to infer the keys⁷ (for instance, if they are provided by the client when creating the dictionary).

In such cases, the type system detects an error: the `replaces` relationship between the two versions defined in the graph implies that clients in version 1 should be able to use all code from version 2 – but without a lens this is not possible.

In such cases, where the developer can not define a lens for a field (e.g. `data`), then the code does not comply with the version graph, as the lens is missing. Intuitively, this indicates that the developer introduced a breaking change from version 1 to version 2, so clients can not migrate automatically. To fix this, the developer must change the version graph and use the `upgrades` relationship between both versions instead.

This typing discipline reflects the nature of a breaking change when evolving class fields, as it prevents the developer from issuing such a change in a replacement release, which, in the absence of a lens, would make the client code crash. Instead, by using the `upgrades` relationship, the developer instructs clients to adapt manually, by migrating to the new version and then type checking their code in that context, correcting manually for any errors.

3.4.2 Method lenses

Similar to field lenses, method lenses map how (possibly different) interfaces of the same method evolve between different, related versions.

Consider again the example in Listing 4, where method `m` is refactored in version 2.1 by renaming parameter `x` to `y`.

In this case, the `replaces` relationship implies that clients in version 2, whose code is written using the interface from that version (i.e. with parameter `x`), should be able to use the new implementation of `m` automatically, without their code breaking:

```
1  @at('2')
2  def client():
3      return C().m(x=...)
```

Since the client code is written against the interface defined at version 2, to use the new implementation, the type system requires that the developer define a method lens at version 2 for method `m` at version 2.1. This lens expresses how the method evolved from one version to the other, so that clients can safely use this new definition without rewriting their code. Later, when projecting the code for version 2, the lenses are used to rewrite method definitions, so that they conform to the interface of version 2 while using the implementation from version 2.1. The following is an example of a method lens that renames argument `x` to `y`, while otherwise preserving the semantics:

⁷ Whether this is possible or not depends on the program's intended semantics.

```

1  @get('1', '2', 'm')
2  def lens_m(self, f: Callable[[C, P], T], x: P) -> T:
3      return f(y=x)

```

 **Listing 5** Method lens to rename a parameter

A method lens is a standard class method, annotated with a `@get` decorator (Listing 3, line 3). The body of each lens function is type-checked in the context of the `<from>` version. The type system ensures that the method (`<name>`) is available in both versions (`<from>` and `<at>`), and that the return type of the lens function (`T'`) matches the return type of method `<name>` in version `<from>`.

A method lens takes a reference to the instance object (first parameter, `self`); a reference to the method definition in version `<from>` (`f`), whose signature matches the type of method `<name>` in version `<at>` (in the example, the signature of `f` corresponds to the type of `m` at version 2.1); and all positional and keyword arguments that method `<name>` takes in version `<at>` (in this case, `x`). The parameter `f` is to aid the developer statically expressing how the calls map between the two versions.

In the above example, `f` is a reference to the definition of `m` in version 2. As such, the developer can express how a method call from a client in version 1 maps to the corresponding method in version 2, in this case by calling `f` and passing `x` as the value to `y`.

Method lenses are a suitable mechanism for modelling common software evolution patterns, such as adding/removing/reordering parameters, changing the type of parameters, and changing a method's return type.

Evolution patterns that mostly concern text manipulation (as opposed to program semantics), are typically considered breaking changes (e.g. changing the name of a parameter in a method). In our setting these patterns are well supported, as it is always possible for the developer to express such a pattern in form of a lens, as hinted in the previous example. If they do so, then these patterns can be applied successfully without introducing breaking changes for clients.

The same approach can be used for methods that evolve semantically, not just textually. Consider the following example, of a method that returns a boolean, and is refactored to return 0 instead of `True` and 1 instead of `False` (the implementation details are omitted here as they are not relevant):

```

1  @at('2')
2  def m(self) -> bool: ...
3  @at('2.1')
4  def m(self) -> int: ...

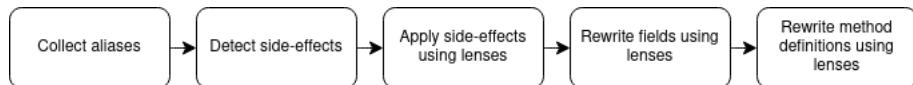
```

Again, the `replaces` relationship between version 2 and 2.1 indicates the developer wants clients in version 2 to use the refactored method introduced in 2.1. As such, the type system requires that they provide a lens expressing how the result of a call to the new implementation, defined in version 2.1, can be mapped to the type of the interface, introduced in version 2. The lens takes as parameter `f` a function that returns an `int`, matching the definition in version 2.1, and returns a value of type `bool`, matching the definition at version 2:

```

1  @get("2", "2.1", "m")
2  def lens_m(self, f: Callable[[C], int]) -> bool:
3      return f() == 0

```



■ **Figure 5** Diagram describing the pipeline of the rewriting procedure.

A client using method `m` in version 2, such as:

```

1 @at('2')
2 def client():
3     return not C().m()
  
```

can use the new code from version 2.1 without refactoring their code, since the mapping from `int` to `bool` (the expected return type for clients in version 2) is provided by the lens.

Similar to field lenses, notice how, when a method evolves semantically, it may not always be possible to devise a lens that models the evolution step, depending on the semantics of the change. Once again, in those cases, to fix the typing error (missing lens), the developer must change the version graph, and define the new version as an upgrade, indicating to client that they must migrate manually and refactor their code to adjust to the new interface.

Method lenses allow clients to use new code without any need for manually refactoring, so developers can express versioning workflows with typical breaking changes, such as changing method signatures, and automatically have clients complying with the versioning policy established in the respective version graph.

3.5 Rewriting procedure

The concept of lenses, described in the previous subsection, allows clients to use code from a version different than the one their code is targeting, even in the presence of different state representations (i.e. class fields) and different method interfaces, respecting the evolution pattern specified by the developer. To do so, we must rewrite the methods that are defined in different versions, so that they conform to the client's version context.

This section describes the procedure to rewrite methods defined at some version `v` so that they match the context of the client version `t`. By doing so, we allow clients to use the new methods without introducing type errors, accounting for side effects and ensuring they are preserved across different version contexts.

The diagram in Figure 5, illustrates the pipeline of the rewriting procedure to rewrite a definition of method `m` from version `v` (where it is defined) to version `t` (which the client is running). Below, we describe each step of this pipeline in detail.

3.5.1 Collecting aliases

In Python, object references are passed by value. When we assign the value of an instance field to a variable, if the variable is mutated, the changes will be reflected back in the instance field. As such, to (statically) detect side-effects on instance fields, we first need to keep track of aliases (to fields) in a given method. Consider the following example of a method `m` at version `v` that appends an element to a list field:

```

1 @at('v')
2 def m(self):
3     x = self.f
4     x.append(1)
  
```

In this example, simple static analysis would not be enough to detect that `self.f` is mutated (since the `append` method is called on a variable, not on a field). So, we need to know that `x` is an alias to a class field (in this case, `f`). To do so, we perform static analysis on assignment statements to collect the aliases of class fields in scope in each method.

From the assignment statement in the above example, we can infer that `x` is an alias to the field `f` of the method caller's class. In some cases, we can also detect if assigning the result of other types of expressions such as a variable, the result of a function call, a list index expression, and so on, also results in a reference to a mutable object field.

We collect all aliases of mutable object fields in a method to use in the next steps, so that we can ensure that the side effects in the code for version `v` are correctly applied when we rewrite it for version `t`.

3.5.2 Detecting side-effects

Now that we have collected all aliases to fields within a method, we can start performing static analysis to detect side-effects. This is crucial to ensure that side-effects on fields of other versions are correctly carried over to the target version, `t`, to which we are rewriting the code. Conversely, detecting cases where no side-effects are produced, avoids having redundant rewrites of such expressions.

The intuition here is the following: we will be using field lenses (in this case, from version `v` to `t`) to rewrite fields (this is detailed further in this section); but, since field lenses are pure functions (i.e. they do not mutate the object calling them), the side effects would be lost if we simply replaced the field by its corresponding lenses – in which case, the side effect would apply to the result of the lens, but not to the current state representation of the class in version `t`.

Consider the following example of method `m` defined at version `v`, where the call to method `pop` will have a side effect on the state of the object (by removing the first element of the list stored in field `f`):

```

1  @at('v')
2  def m(self):
3      x = self.f.pop()
```

We detect side effects to object fields by identifying expressions where the field is passed as a mutable reference to a method⁸ (in this case, method `pop`), so that we can preserve them when rewriting.

To do so, we start by extracting the field to a (new) local variable, rewriting its occurrence in the assignment, and then assigning back to the field the value of the local variable:

```

1  @at('v')
2  def m(self):
3      _x = self.f
4      _x.pop()
5      self.f = _x
```

This logic is expressed in the following rules. In rule (Rw-FIELD-CALL), we rewrite method calls that mutate the calling object. We start by collecting all methods available in the target version `t` ($methods(C, t)$), and selecting those defined at a version other than `t` (i.e.

⁸ It is not always possible to do so by static analysis. For instance, the functions from the Python standard library, such as `pop`, are compiled from C code, which we can not analyse statically. In these cases, we naively assume that function mutates its arguments.

those that need rewriting). Then, for each method, we inspect its body to check if there is a method called on an object ($obj.m'(A) \in body(m)$, where A are the arguments passed in the call). Finally, we check if method m' mutates its caller: this is given by the helper function $mutates(T, m', P_0)$, where T is the object's type and P_0 is the first parameter of method m' (i.e. its caller). If so, we need to rewrite the method call. As shown in the previous example, we start by declaring a new, unused, variable ($x = fresh()$) and assign the object to it ($x = obj$). Then, we call the method on this variable, passing the same arguments rewritten for the context of version t ($x.m'(A')$, where A' is the result of rewriting arguments A). Finally, we assign the value of x back to the object.

In rule (Rw-FIELD-ARGS) (Figure 6), we rewrite method calls that mutate their arguments. Similar to the previous rule, we start by collecting all methods available in the target version t ($methods(C, t)$), and selecting those defined at a version other than t (i.e. those that need rewriting). Then, for each method, we inspect its body to check if there is a method called on an object ($obj.m'(A) \in body(m)$, where A are the arguments passed in the call)⁹. Finally, we select all arguments which are mutated by m' (A'), and we create a fresh variable for each of these arguments (X). To rewrite the call, we start by assigning to each fresh variable the current value of its corresponding argument ($x_i = A'_i$). Then, we call the method, replacing each (mutated) argument with its corresponding variable ($\{A'/x\}$). Finally, we assign to the arguments the value of their corresponding variable (which was mutated in the method call).

$$\begin{array}{c}
 \frac{\begin{array}{c} m \in methods(C, t) \quad v = at(m) \quad v \neq t \\ obj.m'(A) \in body(m) \quad \Gamma \vdash_v obj : T \quad P = parameters(m') \\ mutates(T, m', P_0) \quad A \underset{v \rightsquigarrow t}{\sim} A' \quad x = fresh() \end{array}}{obj.m'(A) \underset{v \rightsquigarrow t}{\sim} x = obj; x.m'(A'); obj = x} \text{ (Rw-FIELD-CALL)} \\[10pt]
 \frac{\begin{array}{c} m \in methods(C, t) \quad v = at(m) \quad v \neq t \quad obj.m'(A) \in body(m) \\ \Gamma \vdash_v obj : T \quad A' = \{ a \in A \mid mutates(T, m', a) \} \\ X = \{ fresh() \mid a' \in A' \} \end{array}}{obj.m'(\bar{a}) \underset{v \rightsquigarrow t}{\sim} \bar{x}_i = \bar{A}'_i; obj.m'(\{A'/x\}); A'_i = x_i} \text{ (Rw-FIELD-ARGS)}
 \end{array}$$

■ **Figure 6** Rules to rewrite fields.

Notice how the code is still typed for the context of version v . This procedure is used whenever object fields (or aliases) are passed as mutable arguments to functions, as described earlier, and also across all language statements, such as loops, return, try-raise, and so on. For example, consider the following example, where an object field is mutated as a condition of an if statement:

```

1  @at('v')
2  def m(self):
3      if self.f.pop():
4          return True

```

To rewrite this statement, we create two new references: one for the object's fields ($_x$, as described earlier); and another for the condition value of the if statement ($_y$). Finally, to

⁹ Note that this rule also applies for functions (e.g. `sort`, which sorts a list in place) and not just methods. For brevity, that case is elided here, although it follows the same logic.

apply the side effects of the method call (`pop`), we assign the value of this reference back to the object's field, before executing the if statement. This ensures the semantics of the (original) code in version `v` are preserved:

```

1  @at('v')
2  def m(self):
3      _x = self.f
4      _y = _x.pop()
5      self.f = _x
6      if _y:
7          return True

```

By the end of this step we should have all side effects to fields expressed as simple assignments of the form `self.f = _v`, where `_v` is the variable holding the value after side-effects are applied.

3.5.3 Rewriting assignments to fields

Following up on the previous step, we now have all side effects expressed as assignments to fields. To rewrite the assignments to the target version `t`, we use the corresponding lenses.

The intuition here is the following: when the developer defines a lens (from `v` to `t`) for a field (`f`), the lens expresses how to compute the value of the field given the state of the object at version `t`. As such, any fields of `v` that appear in the lens will be affected by an assignment to field `f` in the context of version `v`.

Consider the example of method `set_last` (Figure 2a), that assigns a value to field `last`. Since this method is available at version `full`, we must have a way to express how a change to field `last`, in version `init`, affects the state of this version. This is called a *put* lens.

By analysing the lenses from version `init` to version `full` we see that field `last` appears in the lens for field `fname`:

```

1  @get('init', 'full', 'fname')
2  def lens_full(self):
3      return f'{self.first} {self.last}'

```

In practice, this indicates that the value of the field `fname`, in version `full`, is affected by the value of field `last`, in version `init`: if we run the code for this lens, replacing `self.last` with the value that we are assigning to the field, we obtain the matching side effect in field `fname` that results from the assignment. As such, the developer need not provide a definition for a *put* lens, as we can synthesise one from its corresponding *get* lens.

To do so, we add a parameter for each field referenced in the *get* lens, and replace the field reference in the lens body with the (matching) function parameter. The synthesised *put* lens for field `fname` at version `full` is:

```

1  @put('init', 'full', 'fname')
2  def lens_full(self, first, last):
3      return f'{first} {last}'

```

To rewrite an assignment for field `f`, we start by synthesising all necessary *put* lenses. These are synthesised from the corresponding *get* lenses defined at version `v`, for any field (`f'`) from version `t`, that make a reference to field `f` in their body (rule (SYNTH-PUT-LENSSES)). This ensures that a change to field `f`, in version `v`, has its side effects applied to fields `f'` of version `t`.

In the previous example there was only one lens in version `init` using field `last`, so that is the one that is synthesised; in cases where there is more than one, we synthesised all necessary put lenses and unfold the original assignment into multiple assignments, each using a *put* lens for the affected fields. For instance, in Figure 2c, in the lenses from version `full` to version `init`, field `fname` appears in two lenses. This means that both lenses would be needed to correctly apply side effects to field `fname` when rewriting such an assignment to version `init` (which would reflect on fields `first` and `last`, using the same logic).

Now that we have the necessary *put* lenses synthesised, we can use them to rewrite the assignment. For example, to rewrite method `set_last` for version `full`, we can use the synthesised *put* lens (`lens_full`) to apply the side effects resulting from the assignment. To do so, we pass the assigned value (`name`) as the argument to the corresponding field (`last`). To all other unaffected field parameters (i.e. `first`), we pass their current value (in the context of the version where the method is defined, i.e. `self.f` for field `f`).

Finally, to ensure that this code is well-typed in the context of version `t`, we replace all field references (in this case, `self.first`) using the corresponding *get* lens (Figure 2c, line 2), as described in the next subsection. This is expressed in rule (Rw-ASSIGNMENT), where we start by detecting assignments to fields in the method's body ($obj.f = e$), then we rewrite the right-hand side to match the context of version t (e), and finally collect all *put* lenses that affect field f (P). To rewrite the assignment, for each field of t affected by the assignment (F''_i), we assign the result of its corresponding *put* lens (P_i), passing the rewritten value for all unaffected fields ($F_i = F'_i$), and the rewritten assigned value for the assigned field ($f = e'$). The translation of the assignment for version `t` is then:

```
1 def set_last(self, name):
2     self.fname = self.lens_full(first=self.lens_first(), last=name)
```

$$\frac{m \in methods(C, t) \quad v = at(m) \quad obj.f = e \in body(m) \quad \Gamma \vdash_v obj : T \quad e \underset{v}{\rightsquigarrow}_t e' \quad F = \{ f_i \mid f_i \in fields(C, v) \wedge f_i \neq f \} \quad F' = \{ f_i \underset{v}{\rightsquigarrow}_t f'_i \mid f_i \in F \} \quad F'', P = \{ f', put_lenses(T, v, t, f) \mid f' \in fields(C, t) \}}{obj.f = e \underset{v}{\rightsquigarrow}_t obj.F''_i = obj.P_i(F_i = F'_i, f = e')} \text{ (Rw-ASSIGNMENT)}$$

3.5.4 Rewriting field references

As described earlier (section 2), if the method at version `v` contains expressions of references to class fields (e.g. `obj.f`), we need to rewrite these expressions, using the corresponding *get* lens, so they comply with the state of version `t`.

In this step, we do not account for field references to which the previous cases apply as those are already compliant with the target version (i.e. assignments, aliases, and function call arguments).

For example, in Figure 2a, the implementation of method `display` for version `full` is defined in the context of another version, `bugfix`. This definition makes references to fields (`first` and `last`) that are not defined in the context of version `full`. As such, we need to rewrite these references so that they comply with the state of version `full`.

To do so, we replace field occurrences with a call to their corresponding *get* lens provided by the developer (Figure 2c, lines 1-10). This ensures that the code is well-typed in the context of the target version `full`, and that it respects the evolution semantics described by the developer in the implementation of the lenses (rule (Rw-FIELD)):

$$\begin{array}{l}
 1 \quad \text{def display(self):} \\
 2 \quad \quad \text{return f'`\{self.lens_last()\}, {self.lens_first()}`'} \\
 \\
 m \in \text{methods}(C, t) \quad v = \text{at}(m) \quad \text{obj}.f \in \text{body}(m) \\
 \Gamma \vdash_v \text{obj} : T \quad f \in \text{fields}(T, v) \quad l = \text{lens}(T, t, v, f) \\
 \hline
 \text{obj}.f \underset{v \rightsquigarrow_t}{\sim} \text{obj}.l() \quad (\text{Rw-FIELD})
 \end{array}$$

3.5.5 Rewriting method definitions

As described earlier, method lenses allow clients to use new method interfaces and implementations without any need for manually refactoring, allowing developers to express versioning workflows with typical breaking changes and making them non-breaking. This mechanism applies to cases where the method signature or semantics have changed between versions.

Consider again the example in Listing 4 with two implementations of the same method, `m`, where a parameter is renamed from `x` to `y`, a typical breaking change. The semantics of this change is expressed in the lens provided by the developer for this method (Listing 5). With this lens, we can allow the clients of version 2 to use the definition of `m` introduced in version 2.1 without refactoring their code.

To do so, we use the method lens to rewrite the implementation of method `m` when extracting a slice for version 2. The intuition here is that we want to preserve the interface of version 2, since that is what the client code is written against, while using the implementation of version 2.1, which the developer introduced as a replacement for the old definition.

To rewrite the definition using the method lens, we start by adding the new definition of method `m` (from version 2.1) to the program and renaming it, so that we don't introduce a conflict (line 2). Then, we rewrite the body of this method, according to the rewriting procedure described in this subsection, so that it complies with the state of 2. Then, we rewrite the lens function, by removing the parameter `f` (line 4) and replacing it in the body with a reference to the (renamed) method from version 2.1 (line 5). Finally, we rewrite the body of method `m` in version 2 to make a call to the method lens (line 7).

```

1  class C:
2      def __v2_1_m(self, y: int) -> int: ...
3      def lens_m(self, x: int) -> int:
4          return self.__v2_1_m(y=x)
5      def m(self, x: int):
6          return lens_m(x=x)

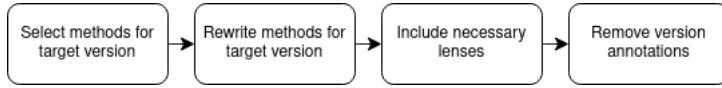
```

The rewriting of method definitions across different versions ensures that clients can write their code against the old interface while taking advantage of the new implementation. This concludes the presentation of the rewriting procedure, which we will use to produce a slice of the program that targets a specific version, as described in the following subsection.

3.6 Program slicing

We propose a slicing procedure, built on top of the rewriting procedure described earlier, applying static program transformations, to extract code for a specific target version from a versioned program. The result should include all code available in that version according to the versioning policy provided by the version graph.

The intuition for the slicing procedure is to be able to project code for a specific release, similar to techniques used in software product lines or variability programming settings.



■ **Figure 7** Diagram describing the pipeline of the slicing procedure.

At its core, the slicing procedure relies on the rewriting procedure to ensure that any code that is reused from other versions is safe to use in the context of the target version t , respecting the evolution semantics specified by the developer in the lenses.

Now that we have defined our rewriting procedure, we can use it to produce a slice for a target version t . In Figure 7, we present a diagram with the pipeline of the slicing procedure. Below, we describe each step of this pipeline in detail:

Select methods. We start by selecting the method definitions (their interface and corresponding implementation, as defined by the lookup policy described earlier) of C that are available at version t , according to the versioning policy of the version graph and the lookup functions described in subsection 3.3. This is expressed in rule (SL-METHODS).

$$\frac{M = \text{methods}(C, t)}{M \subset \text{methods}(\text{slice}(C, t))} \quad (\text{SL-METHODS})$$

Rewrite methods. Given the methods selected in the previous step, we rewrite those that either 1) have an implementation defined at a (base) version other than t or 2) the developer has provided a lens for (i.e. the methods whose semantics have changed). We rewrite these methods using the procedure described earlier so that they match the context of the target version t .

Select lenses. Since the rewriting procedure may need lenses to rewrite expressions for the context of t , we will need to include those (and only those) in the final result. We collect all necessary *get* and *put* lenses (rules (SL-GET-LENSES) and (SL-PUT-LENSES)) that are required for the rewriting procedure, rewrite them to match the context of version t , and include them in the resulting slice for this version. Rule (SL-GET-LENSES) expresses the logic for including *get* lenses: we check each method (m) of the target version (t) that is defined in a different version (v) and, if its body contains a field access expression on an object ($obj.f$) of type T , and a lens is defined between versions v and t of class T for field f ($\text{lens}(T, t, v, f)$), we include it in the final slice of T . Rule (SL-PUT-LENSES) expresses the logic for including *put* lenses: we check each method (m) of the target version (t) that is defined in a different version (v) and, if its body contains a field assignment expression on an object ($obj.f = e$) of type T we iterate over all (*get*) lenses for fields of version t in class T (L), and finally we include the ones where field f is used (L_f).

$$\frac{m \in \text{methods}(C, t) \quad v = \text{at}(m) \quad v \neq t \quad obj.f \in \text{body}(m) \quad \Gamma \vdash_v obj : T \quad l = \text{lens}(T, t, v, f) \quad C' = \text{slice}(T, t)}{l \in \text{methods}(C')} \quad (\text{SL-GET-LENSES})$$

$$\frac{m \in \text{methods}(C, t) \quad v = \text{at}(m) \quad v \neq t \quad obj.f = e \in \text{body}(m) \quad \Gamma \vdash_v obj : T \quad L = \{ \text{lens}(T, v, t, f') \mid f' \in \text{fields}(T, t) \} \quad L_f = \{ l \mid l \in L \wedge \text{self}.f \in l \} \quad C' = \text{slice}(T, t)}{L_f \subset \text{methods}(T')} \quad (\text{SL-PUT-LENSES})$$

$$\frac{l = \text{lens}(C, v, t, f) \quad F = \{ f \mid s \in \text{body}(l) \wedge s = \text{self}.f \} \quad l' = l \{ \text{args} = F \}}{\text{put_lens}(C, v, t, f) = l'} \quad (\text{SYNTH-PUT-LENSES})$$

Remove version annotations. Finally, to produce the slice for version t , we remove any version annotations that may exist, so that the end result is a standard Python program that can be fed to the interpreter to be executed.

In Listing 2, we present the slice of the program in Figure 2a for version `full`. The resulting slice includes the fields and methods of version `full` as defined by the lookup policies described earlier, and any lenses that are necessary to rewrite statements from other versions (e.g. lines 5 and 9).

4 Evaluation

In this section, we empirically evaluate the applicability of our approach by answering the following research questions:

- RQ1.** Can library developers mitigate the occurrence of breaking changes in common evolution patterns?
- RQ2.** Can clients update a library dependency without having to manually refactor their code to account for breaking changes?

4.1 Evaluation design

To setup the evaluation, we started by gathering a set of publicly available Python software libraries with at least two major version releases, v and t . We opted for popular packages, since, given their widespread adoption, these are likely to affect a higher number of clients. We also took care to select packages with different kinds of changes such as renaming methods, fields, or changing method signatures, to better illustrate the applicability of our approach.

For each library, L , we start by defining the version graph. Since we are trying to turn major versions into minor (i.e. so that clients can upgrade transparently without breaking), we define the later version as a replacement of the previous (major) version:

```
1 @version('v')
2 @version('t', replaces=['v'])
```

Then, we select the commit tagged for versions v and t and add the respective version annotations (`at('v')`, `at('t')`) to the methods in each commit. Now that we have the version graph defined and all elements annotated with their corresponding versions, we run the type-checker to detect any missing lenses. We implement the missing lenses, if possible, according to the description stated in the migration guide for L , which corresponds to the semantics the developers intend for each change.

At this stage, we should have a program that type-checks against its version graph (again, if the lenses are possible to implement). Finally, we extract a slice of library L for version v . Now, we can evaluate the applicability of our approach in two ways:

Using client code. In some cases, we were able to use a client program (C) to check if the slice of L for version v conforms to the migration semantics the library developer defined. To do so, we select the commit of C that targets version v of L and type-check this against the slice of L for version v . If the program type-checks, the approach is validated.

Guided by examples in migration guide. As the reader may have understood by now, this evaluation requires a bit of manual labour to setup. This is expected, since the ideas described in this work are more suitable to be applied throughout the development cycle, instead of applied to existing codebases. As a result, it was not feasible to validate some libraries against existing client code. In those cases, we simply used the examples stated in the migration guide (or modelled them ourselves if there are none), and validated the approach in the same way described in the previous point.

4.2 Evaluation results

We conducted our experiments using the libraries listed in Table 1, using client code where possible. The table shows the selected library, the start and target versions we chose, the client used to validate the approach (if any), the number of breaking changes¹⁰, and how many we were able to successfully model.

■ **Table 1** Libraries and clients selected for the experimental evaluation.

Library	Start version	Target version	Client	Changes	Successful
tensorflow	1	2	gpt-2	19	14
emoji	1.7	2	ntfy	1	1
metaapi-python-sdk	22	23	—	2	2
netbox	3.5	3.6	—	5	4
twillio-python	7	8	—	17	14

The following is a summary of the results we obtained for each library:

tensorflow. Out of 19 breaking changes that affected the `gpt-2` package, forcing its developers to migrate manually to support version 2.0 of `tensorflow`, we were able to model 14 successfully. The changes we were unable to model relate to the refactors of `tensforflow` between the two versions, that essentially force the developer to re-structure (and not just rewrite) their code – and our approach does not provide any mechanism for specifying such changes (i.e. clients must always migrate manually). The most relevant example in this case study is the removal of the `tf.multinomial` method. The migration guide points client developers to use another method instead, `tf.random.categorical`. This change can be modelled in our approach by providing a method lens for the `tf.multinomial` method from version 1 to version 2, and implement the lens to use the `tf.random.categorical` method instead.

emoji. Version 2 of the `emoji` package introduces 2 breaking changes, one of which affects the client package `ntfy`. This change involves removing a boolean parameter, `use_aliases`, which defaults to `False`, from method `emojize`. In version 2, client developers should pass `language='alias'` instead of `use_aliases=True`. Our approach is able to model this successfully, by defining a method lens for `emojize` that passes the appropriate value to `language` depending on the value of `use_aliases`. As such, the client package does not need to manually refactor to use the new version.

metaapi-python-sdk. The 2 breaking changes introduced in version 23 of this package involve the rename of a method (`enableMetastatsHourlyTarification` is renamed to `enableMetaStatsApi`), and the rename of a field (`metastatsHourlyTarificationEnabled` is renamed to `metastatsApiEnabled`). Both are supported in our setting and can be successfully implemented, by using a method and a field lens respectively, to allow clients to migrate without refactor.

netbox. Out of the 5 breaking changes introduced in version 3.6, we were able to model 4. The change we were unable to model concerns a dependency (PostgreSQL) that must be upgraded. Since we do not yet support versioning of modules, this is not possible in our setting. The remainder of the changes involve: renaming a field (`device_role` field on the `Device` class is renamed to `role`); changing the name and type of a field

¹⁰When determining the number of breaking changes, we ignored some which fall outside of the scope of this work, particularly when concerning external dependencies.

(field `choices` from the `CustomField` class is renamed to `choice_set`, and its type is changed from a dictionary to `CustomFieldChoiceSets`); removing fields from a class (fields `napalm_driver` and `napalm_args` are removed from the `Platform` class); and changing the return type of a method (reports and scripts are returned within a `results` list). All of these were successfully modelled in our setting.

twillio-python. Out of the 17 breaking changes introduced in version 8, we were able to model 14. The changes we were unable to model concern the renaming of classes: class `ConversationsGrant` is replaced by `VoiceGrant`; and class `IpMessagingGrant` is replaced by `ChatGrant`). We can not model such cases since our type system restricts method lenses (in this case, the constructor method `__init__`) to return the same type as the original definition¹¹. The remainder of the changes involve renaming methods (12 instances) and changing the signature of a method, by removing a parameter (2 instances). All of these were successfully modelled in our setting.

4.3 Evaluation answers

From the results presented in the previous section, we answer the research questions with:

RQ1. Yes, library developers can mitigate (and in some cases, eliminate) the occurrence of breaking changes using our approach.

RQ2. Yes, in most cases clients can update without refactoring their code manually.

5 Related work

Program slicing. In his seminal paper, Weiser [31] describes program slicing as a method for automatically decomposing a program, starting from a subset of its behaviour, and reducing it to a minimal form which still produces that behaviour. This technique is employed in many software engineering activities such as debugging, testing, maintenance and parallelization.

Komondoor et al. [16] propose using slicing to identify duplication in source code, by using program dependence graphs and program slicing to find clones (instances of duplicated code) that are then displayed to the programmer. In the same thread, Gupta et al. [11] suggest a new approach for locating faulty code, with the use of a delta debugging algorithm to identify a minimal failure-inducing input which is then used to compute a forward dynamic slice that is intersected with the statements in the backward dynamic slice of the erroneous output, to compute a failure-inducing chop.

More recently, Maras et al. [18] have applied program slicing to extract the code implementing a certain behaviour for a client-side web application, based on a web page dependency graph. Maruyama et al. [19] propose a slicing mechanism to extract code changes necessary to construct a particular class member of a Java program, based on the history of past code changes which are represented by edit operations recorded on source code of a program, helping programmers avoid replaying edit operations that are non-essential to the construction of class members they are analysing.

To the best of our knowledge, ours is the first attempt to use slicing techniques to handle program variability and versioning.

Update programming. Erwig and Ren [9], Apel and Hutchins [3] introduce an extension to Haskell that supports update programming, where a program is an abstract data type whose building blocks are language terms. They provide a mechanism to script changes in

¹¹This is detailed later on, in section 6.

programs, creating new terms and changing existing ones. Hazelnut [24] is a core calculus that builds on typed “holes” and a gradual type theory that features a type system for expressions with holes and a language of edit actions ensuring that every edit state has static meaning. Both these approaches allows for progressive program construction, as well as giving semantic meaning to incomplete code. We maintain the history of programming versions, well-formed by construction, instead of defining semantics for partial programs [25]. Such history is a guide to the program slicing procedure in VFJ, unlike others where an edit calculus is needed to understand changes ([19]).

Delta-oriented programming. Schaefer et al. [28] introduce DOP, a programming language for designing software product lines based on the concept of program deltas. The implementation of software product lines is divided into a core module, comprising a complete valid product, and a set of delta modules, changes to be applied to the core module to target other products/variations. The language further ensures that all product variations are well typed.

Multiversion systems analysis. The analysis of multiversion systems is usually a project management activity that tries to detect change patterns in the code, and assessing risks of interference between development threads that may result in the introduction of vulnerabilities [14], code repetition [15] and maintenance hurdles [4, 13, 8, 30], and the other difficulties in the management of multiple versions [10, 34, 12, 29]. Our approach acts preventively by detecting illegal evolution steps in the development history and also complements update and delta oriented programming approaches [2, 9, 28] by recording a modification history and allowing (legal) branching in the code base.

6 Limitations and future work

Support for versioned modules. Currently, we do not support versioning of modules. In doing so, we would be able to 1) declared versioned elements at the module level (e.g. functions, constants, variables) and 2) defined versioned imports of packages (i.e. at some version v , we want to import version t of package p). The main challenge is devising a syntax for declaring a module-level version graph (since we use decorators, which are only valid for classes and functions), and devising a syntax for versioned imports.

Structural typing for lenses. Currently, the type system requires that method lenses return the same type (or a subtype) of the original method definition. This forbids us from, for example, defining a lens to rename a class C to D (which would be reflected on the lens of `__init__` method of C , by returning an object of type D). Since our type checker uses nominal sub-typing, this is not possible (since D is not syntactically declared as a subtype of C). As such, we intend to define a structural sub-typing discipline for method lenses.

Inlining for lenses. The rewriting procedure for methods and fields replaces their occurrences with calls to the corresponding lenses. However, from the experiments we conducted, it’s clear that these are, more frequently than not, single line expressions (e.g. when renaming a method argument, or renaming a field). To declutter and optimize the resulting slice, we intend to implement an `@inline` decorator for lenses whose body is a single return statement, to indicate that the lens can be inlined instead of rewriting to a function call.

Version-aware development environment. From a user experience perspective, we believe this approach is not yet suitable for adoption. As such, we intend on implementing tools for a version-aware development environment that would automate most of the common refactoring practices (e.g. moving a method, renaming a field). We are working on an extension for VS-Code to do so, and also plan on extending `rope`, a refactoring library for Python, to account for refactoring of versioned programs. Ideally, the developer would

apply the refactor from the extension in the IDE, and the versioned program would be changed accordingly to include the proposed refactor (for instance, renaming a method would introduce a new definition and its corresponding lens).

7 Conclusions

We build on prior work that presents a language-based approach for a version control system incorporating semantic knowledge of the evolution steps in the code and allowing code sharing and reuse across versions of a software product. We extend it with support for method transformations, and for state and side-effects in an imperative setting.

We instantiate this approach in a large subset of the Python programming language, and demonstrate its applicability by evaluating it against different versions of popular Python packages. We show that this approach is suitable for capturing common software evolution steps, rich versioning workflows, and streamlining the delivery of a snapshot for a given version. We provide a type system to detect conflicts and unintended breaking changes, that operates on a semantic level on top of the entire version graph and its classes, and a slicing compiler to extract the Python code targeting a single version.

References

- 1 ast - Abstract Syntax Trees, 2023. URL: <https://docs.python.org/3/library/ast.html>.
- 2 Edward Amsden, Ryan Newton, and Jeremy Siek. Editing Functional Programs Without Breaking Them. In *IFL 2014*, 2014.
- 3 Sven Apel and Delesley Hutchins. A calculus for uniform feature composition. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(5):1–33, 2008.
- 4 Keith H Bennett and Václav T Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, 2000.
- 5 Luís Carvalho and João Costa Seco. Deep semantic versioning for evolution and variability. In *PPDP 2021*, pages 1–13, 2021.
- 6 Siwei Cui et al. PYInfer: Deep Learning Semantic Type Inference for Python Variables, 2021. arXiv:2106.14316.
- 7 Luca Di Grazia et al. The evolution of type annotations in python: an empirical study. In *ESEC/FSE 2022*, pages 209–220. ACM, 2022.
- 8 S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 2001.
- 9 Martin Erwig and Deling Ren. A rule-based language for programming software updates. In *Proceedings of the 2002 ACM SIGPLAN workshop on Rule-based programming - RULE '02*, Pittsburgh, Pennsylvania, 2002.
- 10 T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 2000.
- 11 Neelam Gupta, Haifeng He, Xiangyu Zhang, and Rajiv Gupta. Locating faulty code using failure-inducing chops. In *Proceedings of the 20th IEEE/ACM International Conference On Automated Software Engineering - ASE '05*, page 263, Long Beach, CA, USA, 2005. ACM Press. doi:10.1145/1101908.1101948.
- 12 P. Hosek and C. Cadar. Safe software updates via multi-version execution. In *2013 35th International Conference on Software Engineering (ICSE)*, May 2013. doi:10.1109/ICSE.2013.6606607.
- 13 C. Izurieta and J. M. Bieman. How Software Designs Decay: A Pilot Study of Pattern Evolution. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, 2007.

- 14 J. Kim, Y. K. Malaiya, and I. Ray. Vulnerability Discovery in Multi-Version Software Systems. In *10th IEEE High Assurance Systems Engineering Symposium (HASE'07)*, 2007.
- 15 Miryung Kim and David Notkin. Program element matching for multi-version program analyses. In *Proceedings of the 2006 international workshop on Mining software repositories - MSR '06*, 2006.
- 16 Raghavan Komondoor and Susan Horwitz. Using Slicing to Identify Duplication in Source Code. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Patrick Cousot, editors, *Static Analysis*, volume 2126, pages 40–56. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. doi:10.1007/3-540-47764-0_3.
- 17 Li Li et al. Scalpel: The python static analysis framework. *arXiv preprint*, 2022. arXiv: 2202.11840.
- 18 Josip Maras, Jan Carlson, and Ivica Crnkovic. Client-side web application slicing. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 504–507, Lawrence, KS, USA, 2011. IEEE. doi:10.1109/ASE.2011.6100110.
- 19 Katsuhisa Maruyama, Eijiro Kitsu, Takayuki Omori, and Shinpei Hayashi. Slicing and replaying code change history. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 246–249. ACM, 2012.
- 20 Stuart McIlroy et al. Fresh apps: an empirical study of frequently-updated mobile apps in the Google play store, 2016.
- 21 Raphaël Monat et al. Static type analysis by abstract interpretation of python programs. In *ECOOP 2020*, 2020.
- 22 Kashif Munawar and Muhammad Shumail Naveed. The impact of language syntax on the complexity of programs: A case study of java and python. *Int. J. Innov. Sci. Technol.*, 4:683–695, 2022.
- 23 Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A Hammer. Live functional programming with typed holes. In *Proceedings of the ACM on Programming Languages*, volume 3, pages 1–32. ACM New York, NY, USA, 2019.
- 24 Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. Live functional programming with typed holes. *Proceedings of the ACM on Programming Languages*, 2019.
- 25 Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. *ACM SIGPLAN Notices*, 2017.
- 26 Tom Preston-Werner. Semantic Versioning 2.0.0, 2023. URL: <https://www.semver.org>.
- 27 S. Raemaekers, A. Van Deursen, and J. Visser. Semantic versioning and impact of breaking changes in the Maven repository. *Journal of Systems and Software*, 129:140–158, 2017.
- 28 Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-Oriented Programming of Software Product Lines. In *Software Product Lines: Going Beyond*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
- 29 Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic software updates: A vm-centric approach. *SIGPLAN Not.*, 44(6):1–12, June 2009. doi:10.1145/1543135.1542478.
- 30 Rick Wash, Emilee Rader, Kami Vaniea, and Michelle Rizor. Out of the loop: How automated software updates cause unintended security consequences. In *10th Symposium On Usable Privacy and Security ({SOUPS} 2014)*, 2014.
- 31 Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.
- 32 Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. Python probabilistic type inference with natural language support. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 607–618. ACM, November 2016.
- 33 Lyuye Zhang et al. Has My Release Disobeyed Semantic Versioning? Static Detection Based on Semantic Differencing, 2022. arXiv:2209.00393.
- 34 Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 2005.

Indirection-Bounded Call Graph Analysis

Madhurima Chakraborty 

University of California, Riverside, CA, USA

Aakash Gnanakumar 

University of California, Riverside, CA, USA

Manu Sridharan  

University of California, Riverside, CA, USA

Anders Møller  

Aarhus University, Denmark

Abstract

Call graphs play a crucial role in analyzing the structure and behavior of programs. For JavaScript and other dynamically typed programming languages, static call graph analysis relies on approximating the possible flow of functions and objects, and producing usable call graphs for large, real-world programs remains challenging.

In this paper, we propose a simple but effective technique that addresses performance issues encountered in call graph generation. We observe via a dynamic analysis that typical JavaScript program code exhibits small levels of indirection of object pointers and higher-order functions. We demonstrate that a widely used analysis algorithm, wave propagation, closely follows the levels of indirections, so that call edges discovered early are more likely to be true positives. By bounding the number of indirections covered by this analysis, in many cases it can find most true-positive call edges in less time. We also show that indirection-bounded analysis can similarly be incorporated into the field-based call graph analysis algorithm ACG.

We have experimentally evaluated the modified wave propagation algorithm on 25 large Node.js-based JavaScript programs. Indirection-bounded analysis on average yields close to a 2X speed-up with only 5% reduction in recall and almost identical precision relative to the baseline analysis, using dynamically generated call graphs for the recall and precision measurements. To demonstrate the robustness of the approach, we also evaluated the modified ACG algorithm on 10 web-based and 4 mobile-based medium sized benchmarks, with similar results.

2012 ACM Subject Classification Theory of computation → Program analysis

Keywords and phrases JavaScript, call graphs, points-to analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.10

Supplementary Material Software: <https://zenodo.org/doi/10.5281/zenodo.12720724>

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

1 Introduction

The construction of accurate call graphs is crucial for various static analysis tasks. Call graphs provide a comprehensive representation of the calling relationships between functions, enabling analysis techniques such as vulnerability and bug detection, program comprehension, and refactorings [7, 20, 18, 3, 29]. Static call graph analyzers aim to over-approximate, meaning that they may include false positives, i.e., unexecutable call edges. Analysis time

 © Madhurima Chakraborty, Aakash Gnanakumar, Manu Sridharan, and Anders Møller;
licensed under Creative Commons License CC-BY 4.0

38th European Conference on Object-Oriented Programming (ECOOP 2024).

Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 10; pp. 10:1–10:22

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

10:2 Indirection-Bounded Call Graph Analysis

tends to correlate with the number of call edges produced, so improving precision can also improve analysis time. Although soundness is desirable, all existing practical whole-program analyses by design sacrifice some amount of soundness to achieve useful precision and scalability [15]. This perhaps makes them unsuitable for code optimization purposes, but other use cases can tolerate false negatives (e.g., many bug finding and vulnerability detection tools aim to expose issues but not to prove their absence). Nevertheless, achieving high accuracy and low analysis time for large, real-world programs is challenging due to the inherent complexity of call graph generation. Analysis time is often prohibitively large, so it is important to explore new approaches that can substantially reduce analysis time even if the price is slightly more false negatives.

As JavaScript has both objects and functions as first-class values, and it has no static type system, constructing call graphs for JavaScript programs generally requires reasoning about the possible flow of objects and functions through the program being analyzed. Functions frequently appear both as arguments and return values of other functions and as values of object fields. State-of-the-art call graph analyzers for JavaScript are based on subset-based flow-insensitive analysis techniques [2]. Objects are typically modeled using allocation-site abstraction [20, 6], or more coarsely using field-based analysis [7, 8]. Functions are tracked using variations of control-flow analysis [17]. In general, the analyses can be expressed using conditional subset constraint systems, which are solved using cubic-time algorithms [21, 25].

Such algorithms build a call graph for a given program iteratively until a fixpoint is reached. Iteration is necessary because of indirections that may occur. For example, if a higher-order function f contains a call $g(\dots)$ to a function that is provided via a parameter g of f , then the call edges for that call site cannot be resolved until the analysis has inferred the calls to f . Similarly, the possible values at an object field read operation $x.a$ generally cannot be obtained until the analysis has inferred which objects x may reference. Different analysis algorithms solve the analysis constraints in different orders, but there is one algorithm, the wave propagation algorithm by Pereira and Berlin [21], that directly follows the levels of indirections, as we explain in detail in Section 4. That algorithm was designed for points-to analysis but is also well suited for call graph analysis.

Interestingly, in real-world code, we observe that function values typically do not flow through many levels of indirection, which means that call graph analysis only needs few iterations to infer most of the possible call edges. In Section 2 we show that creating call graphs for higher-order functions (if not involving objects) requires only as many iterations as the maximum order of the functions, and a similar property holds when objects are involved. Therefore, when wave propagation-style analysis has reached a certain number of iterations, all call edges that are discovered after that point must be false positives that arise only due to analysis imprecision (assuming an idealized sound analysis). Thus, simply by terminating the wave propagation algorithm after a fixed number of iterations, we can reduce analysis time while only missing the relatively few call edges that involve high levels of indirection. This also works when objects are involved; however, due to the asymmetric nature of field read and field write operations (see Section 2) it is beneficial to leave analysis constraints for field read operations and method calls unbounded. This is the first work to utilize observations about low levels of indirection in data flows to achieve a more scalable static analysis.

Another program analysis technique that has proven effective for JavaScript programs is the approximate call graphs (ACG) algorithm of Feldthaus et al. [7]. This algorithm applies field-based analysis, meaning that objects are modeled more abstractly, which generally leads to faster but also less accurate analysis compared to techniques that use allocation-site abstraction of objects. We demonstrate that the ACG algorithm can also easily be adapted to indirection-bounded analysis.

The proposed approach is inspired by the recent work of Mathiasen and Pavlogiannis [16] on the complexity of Andersen’s pointer analysis. One of their key results is that a version of pointer analysis where the number of store operations (corresponding to field write operations in our setting) in witnesses of points-to relations is bounded can be solved in sub-cubic time. They conjecture that the level of indirection of store operations is typically small in practice, but they have left the practical realizations and an experimental evaluation for future work, which we explore here in the context of call graph analysis for JavaScript.

Compared to ad hoc approaches to reduce analysis time, for example, stopping analysis after a time-out, the proposed indirection-bounded approach gives more predictable and interpretable outcomes because of the connection to the program semantics. That is, the results of indirection bounding under a particular bound are deterministic, and one can give a precise semantic characterization of the types of data flows that will be missed due to the bound. Also, instead of tuning time-outs for individual programs, indirection-bound analysis gives good results with a fixed bound applied uniformly for all programs.

In summary, the contributions of this paper are:

- We propose the use of *indirection-bounded* analysis (Section 4) for achieving faster call graph analysis while with little sacrifice in recall, which can be a useful compromise when analyzing large, real-world programs.
- We demonstrate via a dynamic analysis (Section 3) that typical JavaScript programs tend to exhibit small levels of indirections of object pointers and higher-order functions, thereby semantically justifying the use of indirection-bounded analysis.
- By incorporating indirection-bounded analysis into an existing state-of-the-art call graph analyzer for JavaScript that uses the wave propagation algorithm, we present experimental results (Section 5) on 25 large open source programs, showing that the approach on average (geometric mean) results in a 2X speed-up of the analysis with only 5% reduction in recall (and nearly identical precision) relative to the baseline analysis when using dynamically generated call graphs for measuring recall and precision. For many use cases, this can be a valuable trade-off between analysis time and recall. Applying the technique on 10 web-based and 4 mobile-based medium-sized benchmarks using the modified ACG algorithm similarly resulted in an approximately 2X speed-up in analysis time, with only a 1% reduction in recall (again with nearly identical precision).

2 Motivating Examples

In a call graph, call sites and functions are represented as nodes, and a directed edge from node a to node b indicates that call site a may invoke function b at runtime. Sometimes, individual call sites are abstracted by their enclosing function, so that the call edges are from functions to functions.

The accuracy of a call graph construction technique can be measured by its precision and recall relative to the (noncomputable) semantically correct call graph for a given program, or using a call graph produced via one or more dynamic executions of the program as an approximation. (A sound analysis will have perfect recall, but as noted in Section 1, practical analyses are not fully sound.) Different use cases have motivated different metrics in the literature [5]. With the *call site targets* metric [7], precision for a specific call site is computed as the percentage of true (i.e., semantically possible) call targets among those predicted by the static analysis, and recall is the percentage of predicted targets among the true targets. The precision and recall for an entire program are then computed as the averages over all call sites that are semantically reachable. A variant is the *call edge sets* metric [30], which computes precision and recall by comparing the sets of call edges produced by the static

10:4 Indirection-Bounded Call Graph Analysis

```

1 function f1() { }
2 function f2(p1) {
3   p1(); // #5a
4 }
5 function f3(p2) {
6   p2(f1); // #4a
7 }
8 function f4(p3) {
9   p3(f2); // #3a
10}
11function f5(p4) {
12  p4(f3); // #2a
13}
14f5(f4); // #1a

```

(a) Function arguments.

```

15 function g1() { }
16 function g2() { return g1; }
17 function g3() { return g2; }
18 function g4() { return g3; }
19 function g5() { return g4; }
20 var t5 = g5(); // #1b
21 var t4 = t5(); // #2b
22 var t3 = t4(); // #3b
23 var t2 = t3(); // #4b
24 t2(); // #5b

```

(b) Function return values.

Figure 1 Example programs that illustrate indirection levels for function calls.

```

25 var x = {
26   f: function() {}
27 };
28 x.a = x;
29 var f1 = x.f;
30 var f2 = x.a.f;
31 var f3 = x.a.a.f;
32 var f4 = x.a.a.a.f;
33 var f5 = x.a.a.a.a.f;
34 f1(); // #1c
35 f2(); // #2c
36 f3(); // #3c
37 f4(); // #4c
38 f5(); // #5c
39 var x0 = {
40   f: function() {}
41 };
42 x0.f(); // #1d
43 x0.f1 = x0;
44 x0.f1.f(); // #2d
45 var x1 = x0.f1;
46 x1.f2 = x0;
47 x0.f2.f(); // #3d
48 var x2 = x1.f2;
49 x2.f3 = x0;
50 x0.f3.f(); // #4d
51 var x3 = x2.f3;
52 x3.f4 = x0;
53 x0.f4.f(); // #5d
54 var x = {
55   m1: function() { return this; },
56   m2: function() { return this; },
57   m3: function() { return this; },
58   m4: function() { return this; },
59   m5: function() { return this; }
60 };
61 var t1 = x.m1(); // #1e
62 var t2 = t1.m2(); // #2e
63 var t3 = t2.m3(); // #3e
64 var t4 = t3.m4(); // #4e
65 t4.m5(); // #5e

```

(a) Field reads.

(b) Field writes.

(c) Method calls.

Figure 2 Example programs that illustrate indirection levels for object field accesses.

analysis and the dynamic analysis. The *reachable functions* metric [26] and the *reachable edges* metric [10] instead compare the sets of functions or call edges, respectively, that are reachable from the program entry points (e.g., application modules).

Figures 1 and 2 contain five small example JavaScript programs that illustrate the indirections that can arise when computing call graphs. The red edges show the call edges, pointing from call sites to functions. In Figure 1a, function `f5` is a higher-order function, which is called at line 14 with `f4` as argument. The function `f4` is itself a higher-order function that is then called at line 12 with `f3` as argument, etc., until finally line 3 calls `f1`. In other words, `f5` is a 5'th-order function, `f4` is a 4'th-order function, etc. This means that the call edge from the call site marked `#1a` must be discovered before the call edge for `#2a`, etc., until after a total of 5 indirections have been resolved, the call edge for `#5a` can be found. Figure 1b shows a similar example of a 5'th-order function where also 5 levels of indirections arise, but this time due to return values rather than arguments. As we explain in Section 4, analysis algorithms like wave propagation [21], ACG [7], or 0-CFA [24] can compute the call graphs for these programs in 5 iterations. 5'th-order functions are not common in real-world code, which we can exploit to terminate analysis early and save time without risking too many missed call edges (i.e., false negatives).

A similar situation arises when objects and field access operations are involved. In Figure 2a, lines 25–28 set up a simple object structure. The variables `f1`–`f5` all refer to the same function in this case, but via different levels of indirections. Specifically, discovering the call edge for `f5` at call site #5c requires 5 levels of indirection because of the chain of field reads at line 33. Such long chains of field reads operations (sometimes split into smaller parts with variables holding intermediate results) are not uncommon in real-world code, which is why we give special treatment to these operations in the following sections.

On the other hand, it is perhaps less obvious how field write operations can lead to high levels of indirections. Figure 2b shows an example where call site #5d has 5 levels of indirection. That call site cannot be resolved until the analysis has discovered that `x0.f4` is an alias of `x0`. This in turn requires discovering that the field write on the previous line updates `x0.f4`, since `x0` and `x3` are aliases, and so on through each of the aliasing relations established by the preceding lines.¹

Finally, Figure 2c shows an example with chains of method calls, combining objects and functions. Since each method call consists of a pair of a field read and a function call, this example involves a total of 10 levels of indirection to resolve call site #5e. As with chains of field reads, this pattern is also common in real code (e.g., with fluent interfaces [9]), which suggests that method calls should be treated in the same way as field reads in indirection-bounded analysis.

3 Dynamic Indirection Bound Estimation

To confirm the intuitions from Section 2 regarding the depth of indirections encountered in real code, we designed a dynamic analysis to estimate the *minimum indirection bound* required for a static analysis to discover each function call observed during execution. With this dynamic analysis, we can observe the true bound under which function values flow to their invocations in an execution, independent of any static analysis limitation or approximation. If these minimum indirection bounds are observed to typically be low, that provides good evidence that using low indirection bounds in a static analysis will preserve most analysis recall. Here we present the design of the dynamic analysis and give results from a study across a large set of benchmarks.

3.1 Language

The first column of Table 1 defines the types of canonical statements in a core language. (The constraint rules in the second column will be explained in Section 4.1.) The statement types are standard for a flow- and context-insensitive Andersen-style points-to analysis [2] for a JavaScript-like language. A program is a set of functions, each of which contains statements of the types shown in the table (more complex assignments and expressions can be normalized to these forms via temporary variables). We elide details of standard language constructs like conditionals, loops, etc., as they are not relevant given our focus on flow-insensitive static analysis. We assume for simplicity that local variable names are unique across functions.

The values in the language are either object values, written `{}` (like a JavaScript object literal) or (first-class) function values, written `p => ...` (using JavaScript arrow syntax). Without loss of generality we assume every function has one parameter and returns some

¹ One might expect that nested object initialization would yield a high level of indirections via field writes, e.g.: `x = { f: ... }; y = {};` `z = {};` `y.b = x;` `z.a = y;` `z.a.b.f();` But, this code has only *one* level of indirection due to field writes, as objects are copied to the base variables for all field writes without indirection. The indirection level due to field reads is 3, due to the call.

Table 1 Statement types for our analysis and the corresponding static analysis constraint rules (discussed in Section 4).

Statement Type	Constraint Rule
$x = \{ \}_i$	$\{o_i\} \subseteq pt(x)$
$x = p \Rightarrow_i \{ \dots \}$	$\{f_i\} \subseteq pt(x)$
$x = y$	$pt(y) \subseteq pt(x)$
$x = y.f$	$\frac{o_i \in pt(y)}{pt(o_i.f) \subseteq pt(x)}$
$x.f = y$	$\frac{o_i \in pt(x)}{pt(y) \subseteq pt(o_i.f)}$
$x = y(z)$	$\frac{f_i \in pt(y) \quad pt(z) \subseteq pt(p_i) \quad pt(ret_i) \subseteq pt(x)}{pt(z) \subseteq pt(p_i) \quad pt(ret_i) \subseteq pt(x)}$
$\text{return}_i x$	$pt(x) \subseteq pt(ret_i)$

value. The first two statement types respectively allocate a new object or new function value and assign it to a variable; each such statement has a unique label i . An $x = y$ statement copies between variables. For object fields, $x = y.f$ loads field f and $x.f = y$ stores to field f . We assume a JavaScript-like semantics where writing to a non-existent object field f creates f on the object (obviating the need for field declarations). Finally, we have $x = y(z)$ statements for calling functions, and $\text{return } x$ statements for returning values. The label i on each return statement identifies the containing function.

3.2 Dynamic Analysis

Here we present our dynamic analysis to estimate minimum indirection bounds. The analysis does not provide an exact value for these bounds; it may under-estimate due to lack of input coverage or unhandled language features, and it may over-estimate due to an imperfect simulation of the static analysis. Still, we have found its results to be accurate in practice (via manual inspection) and useful for understanding indirection levels in real programs.

Algorithm 1 gives pseudocode for our dynamic analysis. We assume the analysis is implemented via an interface similar to that provided by frameworks like Jalangi [23], where a callback provided by the analysis is invoked before and possibly after the execution of each program statement. In Algorithm 1, the callback is the `HANDLESTMT` procedure. For all statement types, we assume `HANDLESTMT` is invoked before the statement s executes, except for calls $x = y(z)$, where we require the callback both before and after (to respectively handle parameter passing and returns).

The dynamic analysis relies on a function α that given an expression e , first evaluates e to a value v and then returns the allocation site for v (i.e., the label of the statement that allocated v). The dynamic analysis tracks bounds for allocation sites instead of individual dynamic values to match the finite abstraction of values typically used by static call graph builders. For readability, in the remainder of this section we refer to values and their allocation sites interchangeably.

Algorithm 1 Dynamic bounds estimation.

```

1: SELECTIVE: boolean
2:  $V$ : map from variable and value to indirection level bound
3:  $F$ : map from object field and value to indirection level bound
4: procedure HANDLESTMT( $s$ )
5:   match  $s$ 
6:     case  $x = \{ \}_i$  or  $x = p \Rightarrow_i \{ \dots \}$ :
7:        $V[x, i] \leftarrow 0$ 
8:     case  $x = y$ :
9:        $v \leftarrow \alpha(y)$ 
10:       $V[x, v] \leftarrow \min(V[x, v], V[y, v])$ 
11:    case  $\text{return}_i x$ :
12:       $v \leftarrow \alpha(x)$ 
13:       $V[\text{ret}_i, v] \leftarrow \min(V[\text{ret}_i, v], V[x, v])$ 
14:    case before  $x = y(z)$ :
15:       $f_i \leftarrow \alpha(y), v \leftarrow \alpha(z)$ 
16:       $t \leftarrow \max(V[y, f_i] + 1, V[z, v])$ 
17:       $V[p_i, v] \leftarrow \min(V[p_i, v], t)$ 
18:    case after  $x = y(z)$ :
19:       $f_i \leftarrow \alpha(y), v \leftarrow \alpha(\text{ret}_i)$ 
20:       $t \leftarrow \max(V[y, f_i] + 1, V[\text{ret}_i, v])$ 
21:       $V[x, v] \leftarrow \min(V[x, v], t)$ 
22:    case  $x = y.f$ :
23:       $b \leftarrow \alpha(y), v \leftarrow \alpha(y.f)$ 
24:      if SELECTIVE then
25:         $t \leftarrow \max(V[y, b], F[b.f, v])$ 
26:      else
27:         $t \leftarrow \max(V[y, b] + 1, F[b.f, v])$ 
28:       $V[x, v] \leftarrow \min(V[x, v], t)$ 
29:    case  $x.f = y$ :
30:       $b \leftarrow \alpha(x), v \leftarrow \alpha(y)$ 
31:       $t \leftarrow \max(V[x, b] + 1, V[y, v])$ 
32:       $F[b.f, v] \leftarrow \min(F[b.f, v], t)$ 
33:  end match
34: end procedure

```

Algorithm 1 computes two maps, V and F . The V map records for each variable x and value v the minimum observed indirection bound under which v flowed to x in the execution. Retaining only the minimum bound for each variable x and value v makes sense for our use case, as a sound static analysis models all possible data flows, and hence would discover the flow of v to x under that minimum bound. F is similar but is keyed on object fields $b.f$, where b is a value and f is a field name. After the analysis completes, the minimum observed indirection bound for discovering that call $x = y(z)$ invokes function f is simply $V[y, f]$.

We now describe the handling of each type of statement in turn. For a creation of an object or function at allocation site i (line 6), $V[x, i]$ is set to 0, as the flow does not involve any indirections. For an assignment $x = y$ (line 8), where v is the value of y , $V[x, v]$ is set to be the minimum of its current value and $V[y, v]$ (since we aim to find minimum observed indirection bounds). Return statements (line 11) are handled just like assignments, updating the bound for the synthetic ret_i variable for the enclosing function.

10:8 Indirection-Bounded Call Graph Analysis

The next case (line 14), handling parameter passing, is the first involving an indirection, here via a call. Here, f_i is the function value being invoked, and v is the value of the parameter. Recall from the discussion of Figure 1a in Section 2 that to discover data flow into a formal parameter from a call site, the analysis must first discover the data flow of the invoked function to the call; the parameter flow occurs at an increased indirection level. Hence, to discover the parameter data flow from this call, the indirection bound must be at least $V[y, f_i] + 1$. Note that finding the flow also requires discovering that v flows to actual parameter z , so the true bound for this flow is the maximum of these two flows (line 16). Finally, line 17 updates the bound for formal parameter p_i to be the minimum observed thus far. Handling of the return value after a call completes (line 18) is analogous to handling of parameters.

Handling of field reads (line 22) and field writes (line 29) is also similar to that for parameter passing. Here, as discussed in Section 2, the increase in indirection level occurs because the static analysis must first observe the data flow of the relevant object into the base variable of the dereference. For both reads and writes, the base object is named b in the pseudocode, and we add 1 to the bound for the flow of b to the statement in each case (line 27 for reads, line 31 for writes). Recall from Figures 2a and 2b that writing code with a high indirection level for field reads is more natural than doing the same for field writes. Accordingly, the pseudocode has a flag `SELECTIVE` to control whether reads should be treated as bounded when computing estimates. If `SELECTIVE` is true, reads are not treated as bounded, and 1 is *not* added to the bound for the flow of b to the base variable y (see line 25). In our implementation, `SELECTIVE` also controls bounding of method calls; this is not shown in Algorithm 1 since our core language (Table 1) contains only function calls (with no receiver argument), not method calls.

Algorithm 1 may over-estimate the bounds required of a static analysis since it does not propagate information from later assignments to previous calls. Consider this JavaScript example:

```
1 x = function f1() { ... };
2 y = /* some flow with minimum bound 2 yielding f1 */
3 z = y;
4 z();
5 y = x;
```

After Algorithm 1 completes, $V[z, f_1]$ will be 2, since f_1 was initially copied to y via a flow of bound 2 (line 2) and then copied to z . However, due to line 5, there exists a flow of f_1 to y with bound 0. The algorithm updates $V[y, f_1]$ accordingly, but does not propagate this update to $V[z, f_1]$. This issue could be addressed by tracing the execution operations and computing a fixed point over that trace; we used the single-pass approach as we did not observe this over-estimation to occur in practice.

3.3 Study Results

Here we present results of applying our dynamic analysis to a suite of Node.js benchmarks to measure minimum indirection bounds in practice.

Implementation. We implemented the analysis atop the NodeProf framework for Node.js dynamic analysis [28]. For scalability, we separated the analysis into a trace generation phase that runs during program execution, followed by a post-processing phase to compute the bounds. For trace generation, NodeProf does not invoke a single callback for assignments, but instead invokes separate callbacks for reads and writes of both variables and object fields. To adapt Algorithm 1 to this structure, we maintain an additional map S from each value

Table 2 Number of call edges with each minimum bound across all benchmarks, for configurations with SELECTIVE enabled and disabled, respectively.

Configuration	0	1	2	3	4	5	6	7	8	9	10-19	20+
SELECTIVE enabled	40,087	16,560	5,958	2,207	356	29	16	-	-	-	-	-
SELECTIVE disabled	37,589	10,064	5,825	3,610	2,214	1,705	992	442	644	676	1,311	141

v to the bound t for v corresponding to the location (variable or field) from which it was most recently read. Then, t is used when updating the bound at the next variable or field write. So, for a statement $x = y$, the analysis first sees a read of v from y , and it sets $S[v]$ to $V[y, v]$. Then, when handling the subsequent write of v to x , it uses $S[v]$ instead of $V[y, v]$ for the bound update (line 10 in Algorithm 1).

Benchmarks and methodology. We created a suite of 74 Node.js benchmarks for our study, as no standard benchmark suite was available. These were randomly selected among the top 1,000 highest ranked JavaScript projects on GitHub which both had unit tests available and that worked correctly with our dynamic analysis infrastructure and implementation. We exercised each benchmark by running its unit test suite. In total, these runs executed calls at 60,601 distinct call sites.

Results. Table 2 gives the results from our study. We present results for two configurations. The first is our preferred configuration, with SELECTIVE enabled, so reads and method calls are treated as unbounded. The second row gives results when all indirections are bounded. Each column shows how many dynamic call graph edges (from call site to callee function) could be discovered within that bound. The numbers are aggregated across all the benchmarks, for a total of 65,213 call edges (greater than the number of distinct call sites, since some sites invoke different functions on different execution paths).

The results show that in both configurations, most call graph edges can be found within a small bound; more than 57% of edges are discoverable within a bound of 0, i.e., the function data flow involves no indirections. Note, however, that with SELECTIVE enabled, significantly more edges are discoverable within bound 1 (6,496 more than with SELECTIVE disabled), and the long tail of call edges with minimum bound 7 or higher is eliminated. In fact, with SELECTIVE disabled, we discovered calls with a bound as high as 75. This result confirms the intuition from Section 2 that long chains of field reads and method calls can occur regularly in real-world programs, justifying special handling.

Overall, the data from our study provide promising evidence that an indirection-bounded static analysis could discover most true call graph edges within a small bound. For the SELECTIVE configuration, roughly 96% (62,605 / 65,213) of calls are reached within bound 2. These insights guided our static analysis design, described in Section 4.

Examples. For the configuration with SELECTIVE enabled, below is an example of a call that involves 4 levels of indirections, from the `express-react-views` benchmark (heavily simplified for readability). Calls that require even higher indirection bounds are rare, as depicted in Table 2, and are challenging to extract due to their complexity.

10:10 Indirection-Bounded Call Graph Analysis

```

1 // in async.js library
2 function series(tasks) {
3     /*0*/_parallel(eachOfSeries, tasks);
4 }
5 function _parallel(eachfn, tasks) {
6     /*1*/eachfn(tasks, function cb1(task) {
7         /*3*/task(function cb2() {});
8     });
9 }
10 function eachOfSeries(tasks, task_cb) {
11     for (var task of tasks) { /*2*/task_cb(task); }
12 }
13 // in client code
14 /*0*/series([function f(next) { /*4*/next(); }]);

```

The `async.js` library provides a function `series` for running all task callbacks in a provided array. Internally, this functionality is implemented using layers of higher-order functions, leading to the high bound. The calls above are commented `/*0*/` through `/*4*/` to show their bounds. The layered implementation inside `async.js` does enable code reuse within the library, but it leads to convoluted and hard-to-understand control flow, as shown above. As the data in Table 2 show, calls like these requiring bound 4 or greater are quite rare across our benchmarks (only 0.6% of call edges).

In contrast, with `SELECTIVE` disabled, natural code patterns can lead to high bounds, as discussed in Section 2. For example, consider the following code from the `express` benchmark:

```

1 block.paragraph = edit(block._paragraph)
2     .replace('hr', block.hr) /*2*/
3     .replace('heading', '{0,3}#{1,6}+') /*4*/
4     .replace('lheading', '') /*6*/
5     .replace('blockquote', '{0,3}>') /*8*/
6     .replace('fences', '{0,3}({:{3,}|~{3,}})[^\n]*\n') /*10*/
7     .replace('list', '{0,3}({:[*+-]|1[.])}') /*12*/
8     .replace('html', '</?({:tag}({:+|\n|?})|<({:script|pre|style!--})') /*14*/
9     .replace('tag', block._tag) /*16*/
10    .getRegex(); /*18*/

```

This code uses a fluent interface [9]: the `edit` and `replace` methods both return `this`, allowing for chaining of method calls. Each step in the chain involves a field read (to access the method) followed by a call, thereby adding 2 to the minimum bound for this configuration. So, the final call to `getRegex` has 18 levels of indirection. We studied calls with higher bounds and found that they involved a complex mix of field reads and method calls, often spread across multiple functions and files.

4 Indirection-Bounded Call Graph Construction

In this section, we present static call graph construction algorithms that allow for indirection bounding. We first describe a constraint-based formulation of the wave propagation algorithm [21] (Section 4.1), and then present a simplified version for solving the constraints (Section 4.2). In Section 4.3 we extend the algorithm with indirection bounding. Finally, in Section 4.4 we show how further simplifications yield a bounded version of the ACG algorithm of Feldthaus et al. [7]. The static analyses presented in this section work independently of the dynamic analysis presented in Section 3. The purpose of the dynamic analysis was to provide evidence and insights that support the static analysis design by showing how indirection works in different scenarios. This semantic justification helps validate the conclusions drawn from static analysis and ensures the bounding approach is reliable.

4.1 Analysis Formulation

The second column of Table 1 gives constraint rules for computing an Andersen-style points-to analysis for our statement types. The rules are standard; see [25] for a more detailed description. The constraints define what values (objects or functions) must be present in the points-to set of each variable and object field. Given a solution to the constraints, a call graph can be extracted by adding an edge from each call site $x = y(z)$ (or from the function containing the call site) to each function $f_i \in pt(y)$.

Concrete values are abstracted using allocation sites; we write o_i or f_i for an object or function, respectively, allocated at site i . As in the dynamic analysis formulation (Section 3.2), we assume a formal parameter variable p_i and a return variable ret_i for each function f_i . The constraints for field read, field write, and call statements are *conditional* constraints [1] – they each impose new subset constraints based on the contents of another points-to set. For example, the constraint for $x = y(z)$ checks if f_i is present in $pt(y)$, which indicates that $y(z)$ may invoke f_i . In this case, new subset constraints are imposed to capture the data flow from actual parameter z to formal p_i and from the returned value ret_i to x . These conditional constraints correspond directly to the notion of indirections discussed in Sections 2 and 3. Our approach implements indirection bounds by bounding the handling of these conditional constraints, leveraging the structure of the wave propagation algorithm, to be described next.

4.2 Simplified Wave Propagation

Algorithm 2 presents pseudocode for a simplified version of the wave propagation algorithm [21], the basis of our bounding technique. The pseudocode eschews many optimizations critical to the efficiency of the full wave propagation algorithm, including worklists, cycle elimination, and topological sorting. We simplify the pseudocode to clearly expose the two alternating phases of the algorithm, *propagation* and *edge addition*, the aspect of the algorithm most critical to bounding.

Algorithm 2 computes the points-to relation pt using a flow graph G . Each node in G represents a variable or an object field, and each edge $n \rightarrow n'$ in G represents a subset constraint $pt(n) \subseteq pt(n')$ from Table 1. The main entry point is the ANALYZE procedure at line 4.

The algorithm begins with the INIT procedure (line 12), which initializes pt and G based on the simple (non-conditional) constraints for value creation, variable copy, and return statements in Table 1. Then, at lines 6–11, the algorithm alternates between calls to PROPAGATE and ADDEGES until pt and G reach a fixed point. PROPAGATE (line 22) uses a fixed-point loop to ensure that for each edge $n \rightarrow n'$ currently in G , $pt(n) \subseteq pt(n')$. Then, ADDEGES (line 29) processes each field read, field write, and call statement and updates G with new edges based on the current value of pt and the corresponding conditional constraints in Table 1. The clean separation between propagation and edge addition is a key characteristic of wave propagation; it leverages this structure to efficiently eliminate cycles in the constraint graph and compute a topological ordering to minimize propagation work [21].

4.3 Adding Bounds

Given the structure of the wave propagation algorithm, adding bounding of all indirections is straightforward. Algorithm 2 already separates its handling of conditional constraints, which correspond to indirections, into the ADDEGES procedure. So, bounding indirections simply requires limiting the number of times that ADDEGES runs to be less than the bound. We have found that in real implementations that use the wave propagation structure, adding bounding is similarly straightforward.

10:12 Indirection-Bounded Call Graph Analysis

 **Algorithm 2** Simplified wave propagation algorithm.

```

1:  $P$ : program to analyze
2:  $pt$ : points-to relation, initially empty
3:  $G$ : flow graph, initially empty
4: procedure ANALYZE()
5:   INIT()
6:   repeat
7:      $pt' \leftarrow pt$ ,  $G' \leftarrow G$ 
8:     PROPAGATE()
9:     ADDEDGES()
10:    until  $pt' = pt \wedge G' = G$ 
11: end procedure
12: procedure INIT()
13:   for each  $x = \{ \}^i \in P$  do
14:      $pt(x) \leftarrow pt(x) \cup \{ o_i \}$ 
15:   for each  $x = p \Rightarrow_i \{ \dots \} \in P$  do
16:      $pt(x) \leftarrow pt(x) \cup \{ f_i \}$ 
17:   for each  $x = y \in P$  do
18:      $G \leftarrow G \cup \{ y \rightarrow x \}$ 
19:   for each  $\text{return}_i x \in P$  do
20:      $G \leftarrow G \cup \{ x \rightarrow \text{return}_i \}$ 
21: end procedure
22: procedure PROPAGATE()
23:   repeat
24:      $pt' \leftarrow pt$ 
25:     for each edge  $n \rightarrow n'$  in  $G$  do
26:        $pt(n') \leftarrow pt(n') \cup pt(n)$ 
27:     until  $pt' = pt$ 
28: end procedure
29: procedure ADDEDGES()
30:   for each  $x = y.f \in P$ ,  $o_i \in pt(y)$  do
31:      $G \leftarrow G \cup \{ o_i.f \rightarrow x \}$ 
32:   for each  $x.f = y \in P$ ,  $o_i \in pt(x)$  do
33:      $G \leftarrow G \cup \{ y \rightarrow o_i.f \}$ 
34:   for each  $x = y(z) \in P$ ,  $f_i \in pt(y)$  do
35:      $G \leftarrow G \cup \{ z \rightarrow p_i, \text{return}_i \rightarrow x \}$ 
36: end procedure

```

Recall that Section 3.3 showed that field reads often require higher bounds than other indirections, matching the intuition of Section 2. Algorithm 3 is a variant that only bounds indirections via field writes and calls, while leaving handling of field reads unbounded; the changes compared to Algorithm 2 are emphasized with blue (lines 4, 7 and 12). Variables *bound* and *i* are introduced, and only the ADDEDGES procedure of Algorithm 2 is modified. The modified code first adds edges to G based on field reads without checking the bound (lines 5–6). Then, field write and call statements are processed, but only if the field read processing added no new edges to G (the $G' = G$ check on line 7). If handling of reads adds new edges to G , then ADDEDGES returns without incrementing *i*, and another phase of propagation is run (see line 8). Hence, the algorithm only handles stores and calls and increments *i* (lines 8–12) once propagation and edge addition from field reads have iterated to a fixed point.

Algorithm 3 Algorithm 2 modified to bound indirections except for field reads.

```

1: bound: bound on indirections
2: i: current iteration, initially 0
3: procedure ADDEDGES()
4:    $G' \leftarrow G$ 
5:   for each  $x = y.f \in P, o_i \in pt(y)$  do
6:      $G \leftarrow G \cup \{o_i.f \rightarrow x\}$ 
7:   if  $G' = G \wedge i < bound$  then
8:     for each  $x.f = y \in P, o_i \in pt(x)$  do
9:        $G \leftarrow G \cup \{y \rightarrow o_i.f\}$ 
10:    for each  $x = y(z) \in P, f_i \in pt(y)$  do
11:       $G \leftarrow G \cup \{z \rightarrow p_i, ret_i \rightarrow x\}$ 
12:     $i \leftarrow i + 1$ 
13: end procedure

```

The $G' = G$ check on line 7 is crucial for getting the full benefit of unbounded field reads. Consider the following example:

```

1 var y = {...};
2 var x = y.a.b.c;
3 x.m = p => {...};
4 x.m(...);

```

We have three nested field reads at line 2 and one field write on the resulting object at line 3. With unbounded field reads, one would expect the call at line 4 could be discovered with an indirection bound of 1. But, without checking for $G' = G$ at line 7, the counter *i* for writes and calls would still be incremented while handling the reads, exhausting the bound before the relevant data flow from reads was discovered. Algorithm 3 discovers the call with *bound* = 1, as desired. In our implementation, constraints from JavaScript method calls (see Figure 2c in Section 2) are also handled in an unbounded manner, similar to handling of field reads in Algorithm 3.

4.4 Bounded ACG

The ACG algorithm of Feldthaus et al. [7] is a well-known technique for building JavaScript call graphs. ACG uses a field-based modeling of field accesses, unlike the field-sensitive formulation of Table 1. In ACG, reads and writes of object fields are modeled as assignments to and from global variables, and hence they do not introduce indirections for the analysis. Algorithmically, both the original ACG analysis and an indirection-bounded variant can be phrased as a simplified version of wave propagation. Pseudocode for indirection-bounded ACG is given in Algorithm 4; code related to bounding is again shown in blue (lines 12 and 15). The ANALYZE and PROPAGATE procedures (elided) are identical to those in Algorithm 2. Field reads and writes are now handled similarly to assignments in INIT (line 6). ADDEDGES is modified to remove all handling of field accesses. The only remaining indirections to handle in ADDEDGES are calls, as in 0-CFA [24]. Bounding is also simplified compared to Algorithm 3, as field reads do not require any special treatment.

5 Evaluation

We have implemented the techniques of Section 4 in two different analysis frameworks. The first, JELLY [19, 12], implements a field-sensitive call graph analysis using an algorithm like wave propagation [21], and is targeted at Node.js programs. The second, WALA [8], has

 **Algorithm 4** Bounded ACG algorithm [7], as a modified version of Algorithm 2.

```

1: bound: bound on indirections
2: i: current iteration, initially 0
3: procedure INIT()
4:   for each x = p =>i {...} ∈ P do
5:     pt(x) ← pt(x) ∪ {fi}
6:   for each x = y, x = z.y, or z.x = y ∈ P do
7:     G ← G ∪ {y → x}
8:   for each returni x ∈ P do
9:     G ← G ∪ {x → reti}
10: end procedure
11: procedure ADDEGES()
12:   if i < bound then
13:     for each x = y(z) ∈ P, fi ∈ pt(y) do
14:       G ← G ∪ {z → pi, reti → x}
15:     i ← i + 1
16: end procedure

```

an implementation of the ACG algorithm and is targeted at browser-based applications. We modified these implementations to optionally use bounding as described in Sections 4.3 and 4.4.

With these implementations, we performed an experimental evaluation to assess the effectiveness of indirection-bounded call graph construction. We designed our evaluation to answer the following main research questions:

1. How are *analysis running time*, *recall* and *precision* impacted by different values of the indirection bound?
2. How does bounding of field reads and method calls (disabling the SELECTIVE flag of Section 3.2) impact the overall effectiveness of the analysis?

5.1 Benchmarks

For Node.js benchmarks, we further filtered the benchmarks used in the dynamic study (Section 3.3) based on the following three criteria. Initially, we required the dynamic call graph to contain at least 50 call edges to provide a sufficient basis for comparing the precision and recall of the static call graph. Then, we required JELLY could compute a static call graph with a recall of at least 20% as compared to the dynamic call graph; this eliminated some benchmarks where JELLY analyzed only a small portion of the code (typically due to unsupported test frameworks). This criterion also eliminated cases where JELLY ran out of memory. Second, we required that JELLY took at least 15 seconds to analyze the benchmark; for benchmarks that can be analyzed quickly, there is no need for bounding.

This filtering led to a set of 25 benchmarks, whose details are given in Table 3. The benchmarks are large, with thousands of functions and ranging up to more than 9.8MB of code. For each benchmark, the table gives the numbers of packages, modules and functions, the code size, and the analysis time, precision, and recall for JELLY when run without bounding.

For assessing WALA, we constructed a suite of 14 web and mobile benchmarks, shown in Table 4. We included the 10 programs from the TodoMVC suite that were used by Chakraborty et al. in their study [4], and we re-used their test harness to exercise the programs. We also included four sample React Native applications, encompassing the starter

■ **Table 3** Node.js benchmarks used for indirection-bounded static analysis experiments with JELLY.

Benchmark	#Pkgs	#Mods	#Funs	Code Size (kB)	Analysis Time (secs)	Precision (%)	Recall (%)
node-glob	42	186	2,621	1,103	22.38	88.01	90.14
kraken-js	130	303	3,192	1,416	19.62	96.85	37.95
tern	22	233	3,318	1,469	17.67	92.69	80.70
doctoc	96	322	3,212	1,515	15.25	93.97	74.18
js-yaml	59	479	4,440	1,900	28.86	96.42	95.38
react-loadable	55	138	3,192	1,952	405.44	99.35	81.50
babel-plugin-module-resolver	67	537	4,886	2,063	55.55	99.70	91.82
scrape-it	198	390	4,337	2,297	32.80	95.71	80.08
express-react-views	64	489	4,987	2,390	79.19	96.30	98.81
node-oauth2-server	36	262	4,588	2,448	461.66	75.43	95.53
lost	71	994	4,597	2,670	34.28	96.13	98.66
json2csv	112	428	6,743	2,692	29.99	97.24	81.45
homebridge	91	378	5,680	2,787	27.15	92.86	76.51
sharedb	37	266	5,630	3,164	47.64	76.54	65.95
big.js	1	26	1,718	3,306	15.88	96.02	100.00
normalizr	142	747	9,845	3,694	136.44	85.77	81.45
react-refetch	86	408	7,691	3,829	643.23	97.78	90.60
baobab	113	618	9,081	3,957	125.39	92.51	92.79
react-fontawesome	84	410	7,487	4,015	98.99	99.25	99.10
You-Dont-Need-Momentjs	63	742	8,024	5,404	48.10	85.91	84.43
eslint-plugin-compat	80	1,349	7,625	5,513	32.81	97.05	96.20
rewire	103	837	12,316	6,813	82.47	94.62	89.38
bootlint	110	741	10,022	6,835	57.96	96.85	80.46
webpacker	92	922	13,048	6,930	317.69	98.67	83.71
webpack-dashboard	122	1,182	17,783	9,823	834.18	96.31	76.44

app, a to-do app, a chat app, and a bidding app, all gathered from GitHub. We chose apps we could run successfully in a simulator and that worked with our analysis infrastructure, and we manually developed a test harness for each to exercise the code. For these apps, we constructed call graphs for the final shipping version of the code, which is bundled as a single file; hence, package and module counts are omitted in the table. The mobile benchmarks are much larger and more complex than the web benchmarks, explaining the higher analysis times and lower recall for those programs. While some of the smaller benchmarks in this suite can be analyzed very quickly, we retained them in the suite due to the challenge of manually exercising dynamic behaviors in web and mobile apps.

5.2 Experimental Configuration

In our experiments we measure precision and recall using the call site targets metric described in Section 2, i.e., by comparing sets of possible call targets at call sites. We found that this metric most robustly captured the data flows discovered by the static analysis. We also experimented with the reachable functions and reachable edges metrics (see Section 2). These metrics showed similar trends as for the call site targets metric, but are also more fragile since discovery of one additional call graph edge sometimes dramatically impacts overall reachability, making the results harder to interpret.

Table 4 Web and Mobile benchmarks used for experiments with WALA.

Benchmark	#Pkgs	#Mods	#Funs	Code Size (kB)	Analysis Time (secs)	Precision (%)	Recall (%)
Vanillajs	2	8	131	30	0.32	90.13	98.64
Mithril	3	8	136	57	0.35	82.77	90.83
Vue	4	6	623	247	0.68	88.70	95.64
Knockoutjs	4	4	586	308	0.69	89.65	97.31
Jquery	5	5	869	371	2.51	82.75	98.27
Backbone	6	11	950	387	3.93	78.93	97.77
Canjs	5	7	1,105	559	10.15	76.55	97.49
Knockback	8	11	1,798	764	29.25	80.49	96.35
Angularjs	5	9	1,488	1,093	17.08	83.53	95.87
React	5	10	1,876	1,168	45.73	67.67	97.65
Blank-app	–	–	5,203	2,030	309.77	62.33	67.45
Chat-app	–	–	7,119	3,185	1,537.29	55.12	64.85
Bidding-app	–	–	7,073	3,194	1,627.19	55.18	65.02
Todolist-app	–	–	11,678	5,704	8,340.14	57.98	68.28

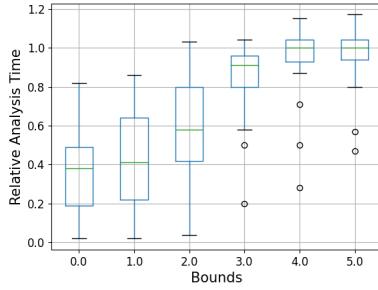
We ran the JELLY experiments on a machine with an 8-core Intel Core i7-11700 processor and 32GB of RAM, running Ubuntu 20.04.6 LTS. For the WALA experiments we used a Google Cloud virtual machine with a 4-core Intel Broadwell Xeon CPU and 64GB RAM running Ubuntu 20.04.6 LTS.

5.3 Results

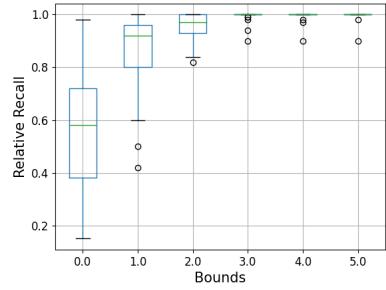
Figure 3 presents our main results for the JELLY experiments. For these experiments, field read and method call indirections were unbounded, the preferred configuration as discussed in Sections 2 and 3.3. The box plots give analysis time and recall relative to the unbounded JELLY analysis. The ideal for a bounded analysis would be to have recall as close to 1.0 as possible, so no recall is lost compared to unbounded, with analysis time as close to 0.0 as possible, maximizing performance. The analysis time data points sometimes extend above 1.0 primarily due to noise in the running time measurements. Additionally, the number of cycle elimination runs in wave propagation [21] can be affected by the use of different types of bounds in the analyzer. This variance can slightly increase or decrease overall analysis running time, depending on the number of cycles in the constraint graph.

Studying Figures 3a and 3b, indirection bound 2 gives the best balance of analysis time improvement and recall. The average analysis time speed-up is roughly 2X (ranging from 0.9X–23.9X), while the average relative recall is 95% (82%–100%) of the unbounded analysis. The high relative recall matches the results of our dynamic study (Section 3.3), where we observed that 96% of dynamic calls were discoverable within a bound of 2. At bound 1, recall loss is significant at 17%, whereas bound 3 shows a minimal recall loss of only 1%, although the analysis time increases significantly. At bound 2, the recall loss is moderate at 5%, offering a balance between recall and analysis time.

Figure 4 gives results for our WALA experiments. Here, we see that bound 1 yields a good overall trade-off, with a roughly 2X average speed-up (1.1X–21.4X) with an average relative recall of 99% (91%–100%). Bound 2 yields 99.99% relative recall on average with a smaller average speedup of 1.6X. We believe the higher recall numbers at lower bounds compared to JELLY are due to the fact that the ACG algorithm has fewer types of indirections

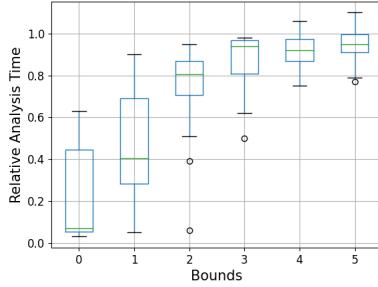


(a) Analysis time.

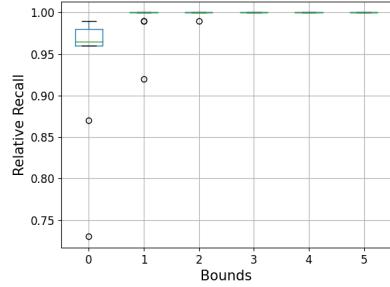


(b) Recall.

■ **Figure 3** Analysis time and recall for JELLY with different indirection bounds and SELECTIVE enabled, relative to JELLY’s unbounded analysis.

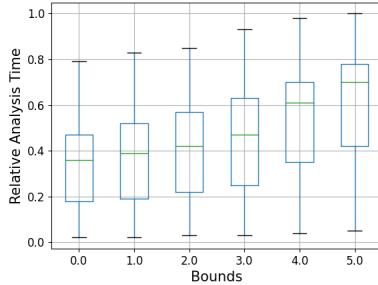


(a) Analysis time.

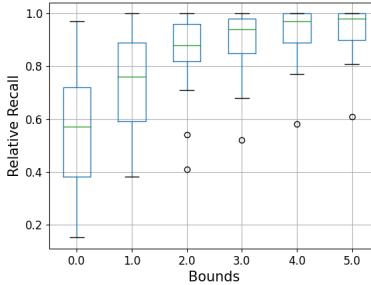


(b) Recall.

■ **Figure 4** Analysis time and recall for WALA with different indirection bounds, relative to WALA’s unbounded analysis.



(a) Analysis time.



(b) Recall.

■ **Figure 5** Relative analysis time and recall for JELLY with reads and methods calls also bounded (SELECTIVE disabled).

to bound (see Section 4.4), and hence more data flow is discovered within a lower bound. For the web benchmarks, the largest observed performance improvements were for the largest benchmarks (e.g., for React we saw a 21X improvement). It would be useful future work to evaluate bounding on a suite of larger web and mobile benchmarks, but exercising such benchmarks to get good coverage of dynamic behaviors can be challenging (due to many user interactions, server-side state, etc.).

Finally, Figure 5 gives data for our second research question, showing results for JELLY with all indirections bounded, including field reads and method calls. Here, the bound with the closest analysis time / recall trade-off to our main configuration is bound 5, with a relative recall distribution fairly similar to bound 2 in Figure 3b. However, at this bound we see some higher outliers in analysis time, ranging up to 89% of the unbounded analysis time. Further, as shown in Section 3.3, with this configuration there is a long tail of calls that require a much higher bound to discover; in Figure 5b, even at bound 5, there is one benchmark with relative recall below 80% and five below 90%. Given these considerations, and the naturalness of code patterns with high numbers of field read and method call indirections (Sections 2 and 3.3), we believe analysis with selective bounding is the better choice for more robust results.

Without indirection bounding, there were 31 benchmarks that could not be analyzed because they caused the analyzer to run out of memory. With indirection bounding, 7 of those benchmarks can be successfully analyzed without running into memory issues.

5.4 Threats to Validity

A threat to the external validity of our evaluation is our choice of benchmarks. For Node.js we chose a large set of realistic benchmarks in a principled manner (see also Section 3.3). For web and mobile we have smaller sets of benchmarks, due to challenges in exercising such benchmarks to collect dynamic data. It is possible that on other types of benchmarks, bounding will be less effective. Our results may also be internally invalid due to bugs in our implementation. We have a variety of regression tests to check correctness of our results and we have done extensive manual inspection of complex examples, reducing this threat. Finally, our choice of call site targets as the precision/recall metric is another threat to external validity. We chose this metric since it best measures the overall effectiveness of the static analysis in capturing function data flows. But, there could be scenarios where a client relies on certain critical edges being present in the static call graph, and those particular edges require a bound greater than 2 to discover. In such cases, a higher indirection bound (or other heuristics) would be required to produce a useful static call graph, leading to higher analysis time.

6 Related Work

The most closely related work is the recent study by Mathiasen and Pavlogiannis [16] on the complexity of different variants of Andersen’s classic pointer analysis. Most importantly, they presented an algorithm for solving Andersen-style pointer analysis instances with bounded numbers of store operations in witnesses of points-to relations, and proved that the algorithm runs in almost quadratic time. In comparison, the indirection-bounded analysis that is based on wave propagation or ACG remains cubic time for a fixed bound. As mentioned in Section 1, Mathiasen and Pavlogiannis conjecture that the level of indirection is typically small but without giving empirical evidence and without experimentally evaluating the effects on analysis time, precision and recall. Furthermore, their work focuses on a C-like language with field-insensitive analysis and without involving higher-order functions, whereas we consider field-sensitive hybrid call graph and pointer analysis for JavaScript. It remains an open problem whether their complexity results can be adapted to field-sensitive analysis. The algorithm and theoretical complexity results by Mathiasen and Pavlogiannis are based on Dyck reachability and matrix multiplications, whereas we base our approach on the wave propagation algorithm that is known to work well in practice. For example, wave propagation is used in the SVF analysis tool for LLVM [27] and also constitutes the core of the PUS constraint solver [14].

Mathiasen and Pavlogiannis [16] additionally showed that their bounded analysis technique is perfectly parallelizable, in contrast to ordinary Andersen pointer analysis. It will be interesting in future work to investigate whether that theoretical property can be exploited in practice to parallelize indirection-bounded call graph analysis.

Horwitz [11] studied a similar notion of levels of pointer indirection, but for reasoning about the analysis precision loss that may occur when normalizing pointer operations, which is not immediately related to bounded analysis techniques.

Uttre and Palsberg [31] introduced a mechanism for analyzing library code only partially to speed up whole-program static analysis of application code. Their technique retains precision but, like indirection-bounded analysis, may lose some recall. We believe such approaches could be combined with indirection-bounded analysis to speed up analysis even further.

Uttre et al. [30] (and follow-up work [13]) propose the notion of a call-graph pruner, which aims to improve analysis precision by eliminating call edges that are likely to be false positives. Like indirection-bounded analysis, that technique may negatively affect recall but in practice often achieves a good balance between precision and recall. The technique works as a post-processing phase and as such does not improve analysis time, and it relies on a learning algorithm that does not provide semantics-based, predictable outcomes.

Bounds have often been used in configuring the abstraction used by a static analysis, e.g., k -limiting for context sensitivity or access path length [25]. Indirection bounding is fundamentally different, in that it heuristically terminates the core fixpoint computation of the analysis before a fixed point is reached. Hence, unlike the aforementioned types of k -limiting, indirection bounding impacts analysis soundness, trading off a small amount of recall for improved scalability.

As pointed out in Section 1 it is well known that practically all whole-program static analyzers have imperfect recall [15], but the analysis results are still useful for many use cases. Reif et al. [22] and Sui et al. [26] investigated this phenomenon empirically for state-of-the-art analyzers for Java, and Antal et al. [3] have made a similar study for JavaScript call graph analysis tools. The more recent work by Chakraborty et al. [4] introduced a method for quantifying the root causes of missing edges in call graphs produced by a field-based static analysis for JavaScript [7]. The dynamic analysis used by Chakraborty et al. inspired the technique presented in Section 3.

7 Conclusion

Indirection-bounded analysis is a simple but effective approach for speeding up call graph analysis while missing relatively few call edges. The approach complements the theoretical results of Mathiasen and Pavlogiannis by providing a practical algorithm and empirical evidence, and it generalizes their bounded mechanism to a language with higher-order functions and to field-sensitive analysis. The results of the dynamic analysis presented in Section 3 indicate that real-world JavaScript code tends to have low levels of indirection of function calls and field writes, which gives a semantic justification of the approach. We have demonstrated that indirection-bounded analysis is straightforward to incorporate into Pereira and Berlin’s wave propagation algorithm and also into the field-based ACG algorithm by Feldthaus et al., and that it can be advantageous to choose a fixed bound independent of the individual programs being analyzed.

For future work, it may be interesting to explore the potential of indirection-bounded analysis for other programming languages, and to investigate whether the parallelizability results of Mathiasen and Pavlogiannis also hold in presence of higher-order functions and field-sensitive analysis.

Data Availability. The supplementary material at <https://zenodo.org/doi/10.5281/zenodo.12720724> contains the benchmarks used in the experimental evaluation and instructions for using indirection-bounded analysis with the open source analysis tools JELLY and WALA.

References

- 1 Alexander Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35(2-3):79–111, 1999.
- 2 Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Copenhagen, 1994.
- 3 Gábor Antal, Péter Hegedüs, Zoltán Herczeg, Gábor Lóki, and Rudolf Ferenc. Is javascript call graph extraction solved yet? A comparative study of static and dynamic tools. *IEEE Access*, 11:25266–25284, 2023. doi:[10.1109/ACCESS.2023.3255984](https://doi.org/10.1109/ACCESS.2023.3255984).
- 4 Madhurima Chakraborty, Renzo Olivares, Manu Sridharan, and Behnaz Hassanshahi. Automatic root cause quantification for missing edges in JavaScript call graphs. In *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany*, volume 222 of *LIPICS*, pages 3:1–3:28. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:[10.4230/LIPIcs.ECOOP.2022.3](https://doi.org/10.4230/LIPIcs.ECOOP.2022.3).
- 5 Madhurima Chakraborty, Renzo Olivares, Manu Sridharan, and Behnaz Hassanshahi. Automatic Root Cause Quantification for Missing Edges in JavaScript Call Graphs (Extended Version). *CoRR*, 2022. URL: <https://arxiv.org/abs/2205.06780>.
- 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. doi:[10.1145/93542.93585](https://doi.org/10.1145/93542.93585).
- 7 Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Efficient construction of approximate call graphs for JavaScript IDE services. In *35th International Conference on Software Engineering, ICSE ’13, San Francisco, CA, USA, May 18-26, 2013*, pages 752–761. IEEE Computer Society, 2013. doi:[10.1109/ICSE.2013.6606621](https://doi.org/10.1109/ICSE.2013.6606621).
- 8 Stephen Fink et al. WALA. <https://github.com/wala/WALA>, 2024.
- 9 Martin Fowler. FluentInterface. <https://www.martinfowler.com/bliki/FluentInterface.html>, 2005. Accessed: 2023-09-24.
- 10 Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. Saving the world wide web from vulnerable JavaScript. In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, pages 177–187. ACM, 2011. doi:[10.1145/2001420.2001442](https://doi.org/10.1145/2001420.2001442).
- 11 Susan Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Trans. Program. Lang. Syst.*, 19(1):1–6, 1997. doi:[10.1145/239912.239913](https://doi.org/10.1145/239912.239913).
- 12 Mathias Rud Laursen, Wenyuan Xu, and Anders Møller. Reducing static analysis unsoundness with approximate interpretation. *Proceedings of the ACM on Programming Languages (PACMPL)*, 4(PLDI):194:1–194:24, 2024.
- 13 Thanh Le-Cong, Hong Jin Kang, Truong Giang Nguyen, Stefanus Agus Haryono, David Lo, Xuan-Bach Dinh Le, and Huynh Quyet Thang. AutoPruner: transformer-based call graph pruning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 520–532. ACM, 2022. doi:[10.1145/3540250.3549175](https://doi.org/10.1145/3540250.3549175).
- 14 Peiming Liu, Yanze Li, Bradley Swain, and Jeff Huang. PUS: A fast and highly efficient solver for inclusion-based pointer analysis. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1781–1792. ACM, 2022. doi:[10.1145/3510003.3510075](https://doi.org/10.1145/3510003.3510075).

- 15 Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundness: a manifesto. *Communications of the ACM*, 58(2):44–46, 2015. doi:10.1145/2644805.
- 16 Anders Alnor Mathiasen and Andreas Pavlogiannis. The fine-grained and parallel complexity of Andersen’s pointer analysis. *Proc. ACM Program. Lang.*, 5(POPL):1–29, 2021. doi:10.1145/3434315.
- 17 Jan Midtgård. Control-flow analysis of functional programs. *ACM Comput. Surv.*, 44(3):10:1–10:33, 2012. doi:10.1145/2187671.2187672.
- 18 Anders Møller, Benjamin Barslev Nielsen, and Martin Toldam Torp. Detecting locations in JavaScript programs affected by breaking library changes. *Proc. ACM Program. Lang.*, 4(OOPSLA):187:1–187:25, 2020. doi:10.1145/3428255.
- 19 Anders Møller and Oskar Haarklou Veileborg. Jelly. <https://github.com/cs-au-dk/jelly>, 2024.
- 20 Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. Modular call graph construction for security scanning of Node.js applications. In *ISSTA ’21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, pages 29–41, 2021. doi:10.1145/3460319.3464836.
- 21 Fernando Magno Quintão Pereira and Daniel Berlin. Wave propagation and deep propagation for pointer analysis. In *Proceedings of the CGO 2009, The Seventh International Symposium on Code Generation and Optimization, Seattle, Washington, USA, March 22-25, 2009*, pages 126–135. IEEE Computer Society, 2009. doi:10.1109/CGO.2009.9.
- 22 Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, and Mira Mezini. Judge: identifying, understanding, and evaluating sources of unsoundness in call graphs. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, pages 251–261. ACM, 2019. doi:10.1145/3293882.3330555.
- 23 Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE’13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 488–498. ACM, 2013. doi:10.1145/2491411.2491447.
- 24 Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie Mellon University, 1991.
- 25 Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. Alias analysis for object-oriented programs. In David Clarke, Tobias Wrigstad, and James Noble, editors, *Aliasing in Object-Oriented Programming*. Springer, 2013. doi:10.1007/978-3-642-36946-9_8.
- 26 Li Sui, Jens Dietrich, Amjad Tahir, and George Fourtounis. On the recall of static call graph construction in practice. In *ICSE ’20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 1049–1060. ACM, 2020. doi:10.1145/3377811.3380441.
- 27 Yulei Sui and Jingling Xue. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 265–266. ACM, 2016. doi:10.1145/2892208.2892235.
- 28 Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. Efficient dynamic analysis for Node.js. In Christophe Dubach and Jingling Xue, editors, *Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria*, pages 196–206. ACM, 2018. doi:10.1145/3178372.3179527.
- 29 Kwangwon Sun and Sukyoung Ryu. Analysis of JavaScript programs: Challenges and research trends. *ACM Comput. Surv.*, 50(4):59:1–59:34, 2017. doi:10.1145/3106741.

10:22 Indirection-Bounded Call Graph Analysis

- 30 Akshay Utture, Shuyang Liu, Christian Gram Kalhauge, and Jens Palsberg. Striking a balance: Pruning false-positives from static call graphs. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 2043–2055. ACM, 2022. doi:10.1145/3510003.3510166.
- 31 Akshay Utture and Jens Palsberg. Fast and precise application code analysis using a partial library. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 934–945. ACM, 2022. doi:10.1145/3510003.3510046.

Regrading Policies for Flexible Information Flow Control in Session-Typed Concurrency

Farzaneh Derakhshan 

Illinois Institute of Technology, Chicago, IL, USA

Stephanie Balzer 

Carnegie Mellon University, Pittsburgh, PA, USA

Yue Yao 

Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

Noninterference guarantees that an attacker cannot infer secrets by interacting with a program. Information flow control (IFC) type systems assert noninterference by tracking the level of information learned (pc) and disallowing communication to entities of lesser or unrelated level than the pc . Control flow constructs such as loops are at odds with this pattern because they necessitate downgrading the pc upon recursion to be practical. In a concurrent setting, however, downgrading is not generally safe. This paper utilizes *session types* to track the flow of information and contributes an IFC type system for message-passing concurrent processes that allows downgrading the pc upon recursion. To make downgrading safe, the paper introduces *regarding policies*. Regarding policies are expressed in terms of integrity labels, which are also key to safe composition of entities with different regarding policies. The paper develops the type system and proves *progress-sensitive noninterference* for well-typed processes, ruling out timing attacks that exploit the relative order of messages. The type system has been implemented in a type checker, which supports security-polymorphic processes.

2012 ACM Subject Classification Theory of computation → Linear logic; Security and privacy → Logic and verification; Theory of computation → Process calculi

Keywords and phrases Regarding policies, session types, progress-sensitive noninterference

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.11

Related Version Technical Report: <https://doi.org/10.48550/arXiv.2407.20410>

Supplementary Material Software (ECOOP 2024 Artifact Evaluation approved artifact):
<https://doi.org/10.4230/DARTS.10.2.4>

Funding *Stephanie Balzer:* Supported in part by the Air Force Office of Scientific Research under award number FA9550-21-1-0385 (Tristan Nguyen, program manager). Any opinions, findings and conclusions or recommendations expressed here are those of the author(s) and do not necessarily reflect the views of the U.S. Department of Defense.

1 Introduction

With the emergence of new applications, such as Internet of Things and cloud computing, today's software landscape has become increasingly *concurrent*. A dominant computation model adopted by such applications is *message passing*, where several concurrently running processes connected by channels exchange messages. A further common aspect is the need for security, ensuring that confidential information is not leaked to a (malevolent) observer.

Information flow control (IFC) type systems [36, 39, 42] rule out information leakage by type checking. These systems statically track the level of information learned by an entity and disallow propagation to parties of lesser or unrelated levels, given a security lattice. The ultimate property to be asserted by an IFC type system is noninterference, a program equivalence statement up to the confidentiality level of an observer. The gold standard is *progress-sensitive noninterference* (PSNI) [24], which treats divergence as an

 © Farzaneh Derakhshan, Stephanie Balzer, and Yue Yao;
licensed under Creative Commons License CC-BY 4.0
38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 11; pp. 11:1–11:29



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

11.2 Regrading Policies for Flexible IFC in Session-Typed Concurrency

observable outcome. PSNI thus only equates a divergent program with another diverging one, whereas *progress-insensitive noninterference* (PINI) regards divergence to be equal to any outcome. Especially in a concurrent setting, PSNI is a sine qua non because the *termination channel* [36] can be scaled to many parallel computations, each leaking “just” one bit [4, 40].

Guaranteeing PSNI, or even PINI for that matter, can become both a blessing and a curse in a concurrent setting. To ensure such a strong property, IFC type systems have to be very restrictive. The troublemakers, in particular, are control flow constructs, such as loops and if statements. Whereas IFC type systems for sequential languages allow the `pc` label¹ to be lowered to its previous level for the continuation of a control flow construct, even if the construct itself runs at high, this treatment is no longer safe in a concurrent setting [39]. To uphold noninterference, IFC type systems for concurrent languages typically forbid high-security loop guards and may even put restrictions on if statements, depending on thread scheduling and attacker model [35, 37, 39].

The use of linearity provides some relief [7, 20, 46–48], allowing high-security loop guards. Linearity also facilitates race freedom, key to guaranteeing observational determinism and, thus, the absence of certain timing attacks [7, 20, 48]. A family of concurrent languages that employ linearity are *session types* [9, 26, 27, 30, 43]. Session types are used for message-passing concurrency, typically in the context of process calculi, where concurrently running processes communicate along channels. A distinguishing characteristic of session types is their ability to assert *protocol adherence*. A session-typed channel prescribes not only the types of values that can be transported over the channel but also their relative sequencing.

In this paper, we develop a flow-sensitive IFC session type system that not only supports recursive processes with arbitrary recursion guards, including high-security ones, but also identifies synchronization patterns that make it safe for the process body to downgrade to the initial `pc` level upon recursion. We refer to this adjustment of confidentiality level as *regrading*. To enforce the safety of regrading, we complement confidentiality with *integrity* [8]. Integrity allows prescribing a process a *regrading policy*, ensuring that any confidential information learned during the high-security parts of the loop cannot be rolled forward to the next iteration. Processes are *polymorphic* in the confidentiality and integrity labels, ensuring maximal flexibility of the IFC type system.

We contribute a type checker for our IFC type system, yielding the language **SINTEGRITY**. The type checker supports security-polymorphic processes using local *security theories*. Well-typed processes in **SINTEGRITY** enjoy PSNI. To prove this result, we develop a *logical relation* for integrity, showing that well-typed processes are self-related (fundamental theorem, Thm. 1). We then prove that the logical relation is closed under parallel composition and that related processes are bisimilar (adequacy theorem, Thm. 3).

Regrading is related to robust declassification [6, 15, 33, 44, 45, 49], as both allow downgrading the `pc` using integrity. In contrast to declassification, which deliberately releases information and thus intentionally weakens noninterference, regrading preserves noninterference. The distinction also manifests in how integrity is used. Whereas integrity is used in robust declassification to convey how trustworthy the information is on which a regrading decision is based, integrity in our work is used to impose extra synchronization policies on regrading processes to prevent leakage by downgrading the `pc` upon recursion. As such, regrading constitutes a more permissive IFC mechanism.

¹ The `pc` (program counter) label approximates the level of confidential information learned up to the current execution point.

Contributions

- The notion of a regrading policy to downgrade a process' confidentiality, retaining PSNI.
- The language **SINTEGRITY**, a flow-sensitive IFC session type system for asynchronous message-passing with confidentiality and integrity to support regrading policies.
- A logical relation for integrity to prove that **SINTEGRITY** processes satisfy PSNI.
- A type checker for **SINTEGRITY**, available as an artifact.

The complete formalization with proofs is available as a technical report (TR) [21].

2 Motivating example and background

This section provides an introduction to session-typed programming and IFC control based on a running example. Our language **SINTEGRITY** is an *intuitionistic linear session type* language [9, 41]; thus, our presentation is specific to that family of session types.

We use a simple bank survey as an example. The survey is carried out by an analyzer at a bank to decide whether to buy or sell a share of stock. The analyzer's decision depends on the opinion of two groups of participants, queried by two surveyors, and a strategy provided by a tactician. For simplicity, we assume that each group of participants only consists of one participant, and the surveyors simply pass the opinion of their participant to the analyzer.

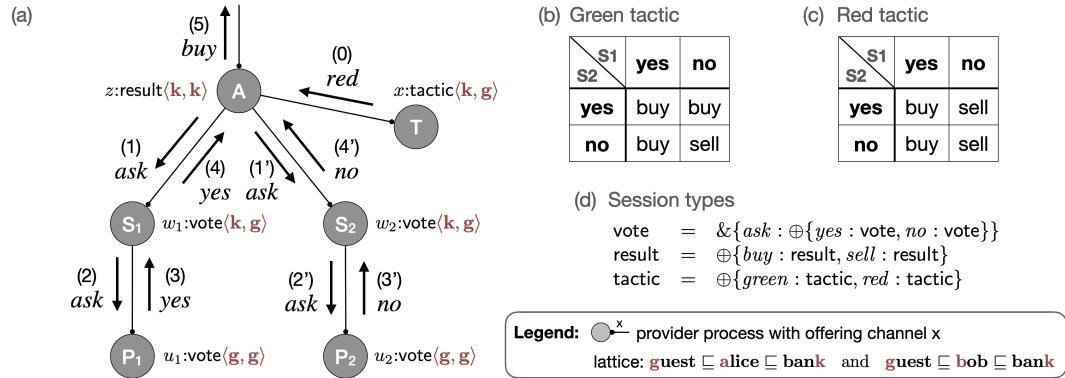


Figure 1 Bank survey: (a) process configuration, (b)/(c) red/green tactic, (d) session types.

A runtime configuration of processes for this example is shown in Fig. 1(a): the analyzer process A, the tactician process T, the surveyor processes S₁ and S₂, along with their participant processes P₁ and P₂, resp. The processes are connected by the channels u₁, u₂, w₁, w₂, x, and z. The figure shows the communications between the analyzer, surveyors, and participants along these channels, with arrows indicating the message being exchanged. The analyzer sends the message *ask* to surveyor S₁ to request a poll (1). Surveyor S₁ then sends the message *ask* to participant P₁ to get their opinion about buying a share (2). Once the surveyor receives P₁'s vote (i.e., either *yes* or *no*) (3), it relays the vote back to the analyzer (4). The analogous communication pattern is repeated between the analyzer and surveyor S₂ and participant P₂ (1'-4'). The final decision whether to *buy* or *sell* (5) of the analyzer is based on the tactic provided by the tactician. For simplicity, we assume that the tactician chooses either a *green* or *red* tactic (0). In the green tactic, the analyzer decides to buy the share if at least one of the surveyors votes yes. In the red tactic, the analyzer buys the stock if the first surveyor votes to buy, regardless of the opinion of the second one (see Fig. 1(b-c)).

Table 1 SINTEGRITY constructs. Upper half: types and terms (before and after exchange), operational meaning, and polarity. Lower half: spawn and forward terms and operational meaning.

Session type (b/a)	Process term (b/a)	Description
$x : \oplus \{\ell : A_\ell\}_{\ell \in L}$	$x : A_k \quad x.k; P$	P provider sends label k along x , continues with P
	$\text{case } x(\ell \Rightarrow Q_\ell)_{\ell \in L}$	Q_k client receives label k along x , continues with Q_k
$x : \& \{\ell : A_\ell\}_{\ell \in L}$	$x : A_k \quad \text{case } x(\ell \Rightarrow P_\ell)_{\ell \in L} \quad P_k$	provider receives label k along x , continues with P_k
	$x.k; Q$	client sends label k along x , continues with Q
$x : A \otimes B$	$x : B \quad \text{send } y x; P$	P provider sends channel $y : A$ along x , continues with P
	$z \leftarrow \text{recv } x; Q_z$	Q_y client receives channel $y : A$ along x , continues with Q_y
$x : A \multimap B$	$x : B \quad z \leftarrow \text{recv } x; P_z$	P_y provider receives channel $y : A$ along x , continues with P_y
	$\text{send } y x; Q$	Q client sends channel $y : A$ along x , continues with Q
$x : 1$	$- \quad \text{close } x$	- provider sends “close” along x and terminates
	$\text{wait } x; Q$	Q client receives “close” along x , continues with Q
$x : Y$	$x : A \quad -$	- recursive type definition $Y = A$ (Y occurs in A)
Judgmental rules		
$(x^{(c,e)} \leftarrow X[\gamma] \leftarrow \Delta) @ \langle c_0, e_0 \rangle; Q_x$		spawn X along $x^{(c,e)}$ with arguments Δ , substitution γ , and running security $\langle c_0, e_0 \rangle$, then continue with Q_x
$x \leftarrow y$		forward $x : A$ to $y : A$

The protocols for these communications can be specified by the session types shown in Fig. 1(d), using the connectives of Table 1. The connectives are drawn from intuitionistic linear logic and obey the following grammar:

$A, B ::= \oplus \{\ell : A_\ell\}_{\ell \in L} \mid \& \{\ell : A_\ell\}_{\ell \in L} \mid A \otimes B \mid A \multimap B \mid 1 \mid Y$,
where L ranges over finite sets of labels denoted by ℓ and k , amounting to primitive values in our system. Type variable Y is a fixed point whose definition $Y = A$ is given in a global signature Σ . The latter is used to define general recursive types. Recursive types must be *contractive* [23], demanding a message exchange before recurring, and *equi-recursive* [19], avoiding explicit (un)fold messages and relating types up to their unfolding. All three types *vote*, *result*, and *tactic* are recursive.

Table 1 provides an overview of SINTEGRITY types and terms. A crucial characteristic of session-typed processes is that a process *changes* its type along with the messages it exchanges. A process’ type therefore always reflects the current protocol state. Table 1 lists state transitions caused by a message exchange in columns 1 and 2 with corresponding process terms in columns 3 and 4. Column 5 describes the computational behavior of a type.

Linearity ensures that every channel connects exactly two processes, thus imposing a *tree* structure on a configuration of processes, as witnessed by Fig. 1(a). We adopt a form of session types corresponding with intuitionistic linear logic, which moreover introduces a distinction between the two processes connected by a channel, identifying one as the *parent* and the other as the *child*, turning the configuration into a *rooted tree*. The parent and child processes have mutually dual perspectives on the protocol of their connecting channel: The child has the perspective of the *provider* and the parent that of a *client*. Column 5 of Table 1 describes the perspective of the client and provider for each type. We assign a polarity to each session type which determines whether the type has a sending semantics or a receiving semantics. For positive types, the provider sends, and the client receives; for negative types, the provider receives, and the client sends. The types with positive polarity are $\oplus \{\ell : A_\ell\}_{\ell \in L}$, $A \otimes B$, and 1 , and the types with negative polarity are $\& \{\ell : A_\ell\}_{\ell \in L}$ and $A \multimap B$.

Figure 2 Secure process implementations of analyzer and surveyor (see Fig. 1), accepted by SINTEGRITY but rejected by existing IFC session type systems

Each process in a configuration is uniquely identified by the channel that connects it to its parent, which we also refer to as its *offering* (or *providing*) channel. We consider the session type of a process to be the protocol of its offering channel. For example, the participant process P_1 in Fig. 1 has type *vote*, which is also the type of the process' offering channel u_1 that connects P_1 to its client S_1 . We say that the client S_1 *uses* the channel u_1 .

The connectives \otimes and \multimap , not used in the example, allow sending channels along channels. Such higher-order channels change the connectivity structure of a configuration: from the perspective of the provider, \otimes turns a child into a sibling and \multimap a sibling into a child. The former is achieved by sending a subtree to the parent and the latter by receiving a subtree from the parent. § 5.4 showcases an example that uses higher-order channels.

It is now time to explore how to implement the processes of our bank survey example. Fig. 2 gives the process definitions of the analyzer and surveyor. A process definition consists of the process signature (first two lines) and body (after =). The first line indicates the typing of channel variables used by the process (left of \vdash) and the type of the providing channel variable (right of \vdash). The former are going to be child nodes of the process. The second line binds the channel variables. In **SINTEGRITY**, \leftarrow generally denotes variable bindings. The channels and the process definitions are annotated with confidentiality and integrity levels (e.g., `<bank, guest>` and `@(guest, guest)`). We will later describe the meaning of these annotations; the reader can safely ignore them for now.

The analyzer first waits to receive a tactic from the tactician along channel x . In either branch (i.e., *green* or *red*), the analyzer proceeds by requesting a vote from surveyors S_1 and S_2 , after which it communicates its decision along its offering channel z before recurring. We remark that the notation $z \leftarrow A \leftarrow w_1, w_2, x$ used for a tail call does not precisely match up with Table 1 because we are deferring a discussion of security annotations and substitutions for security-polymorphic processes to § 5.1. Moreover, a tail call is syntactic sugar for a spawn combined with a forward; i.e., $z \leftarrow A \leftarrow w_1, w_2, x$ desugars to $z' \leftarrow A \leftarrow w_1, w_2, x; z \leftarrow z'$.

We implement a surveyor by two processes S and S' to take advantage of **SINTEGRITY**'s support for regrading, as we will detail in § 3.1. The surveyor starts out as process S and calls process S' right after having received the request from its parent, the analyzer.

Suppose that the tactic is a secret that a participant shall not deduce. The implementations in Fig. 2 respect this security condition: the analyzer interacts with the participants via the surveyors the same regardless of the tactic it received. Existing IFC session type systems [7, 20], however, reject these implementations, because they view the analyzer as

tainted as soon as it learns the secret tactic, and disallow further communication with the participants via the surveyor. This paper relaxes this restriction – while preserving PSNI – and allows the tainted surveyor to interact with the participants while putting safeguards in place (synchronization patterns, § 3.2–§ 3.3 and § 5.2) that prevent the surveyor from leaking the tactic.

3 Key ideas

This section develops the main ideas underlying our flexible IFC session type system; the type system and dynamics is given in § 5. The latter is asynchronous, i.e., non-blocking sends and blocking receives (see § 4.2 and § 5.3). An asynchronous semantics allows for a more permissive noninterference statement since message receipt is not observable.

It may be helpful to foreshadow our attacker model (detailed in § 6.1). We assume that an attacker knows the implementation of all processes and can observe messages sent over channels with lower or equal confidentiality level than the attacker. The attacker cannot measure time but can observe the relative order in which messages are sent along different observable channels. As we aim for PSNI, we need to ensure that an attacker is unable to deduce any information from non-reactiveness either.

3.1 Regrading confidentiality

It is now time to consider the red annotations $\langle c, e \rangle$ on channels and the green annotations $@\langle c_0, e_0 \rangle$ on process terms in Fig. 2, where c, d, c_0 , and e_0 range over levels in the security lattice $\text{guest} \sqsubseteq \text{alice} \sqsubseteq \text{bank}$ and $\text{guest} \sqsubseteq \text{bob} \sqsubseteq \text{bank}$. We focus on the first components c and c_0 for now, which denote confidentiality labels. They are adopted from existing IFC session type systems [7, 20], which are based solely on confidentiality.

The first component c of the pair $\langle c, e \rangle$ indicates the *maximal confidentiality* of a process, i.e., the maximal level of secret information the process may ever obtain. As to be expected, the analyzer (A), the tactician (T), and both surveyors (S_1 and S_2) have maximal confidentiality bank , as they are affiliated with the bank and have the clearance of knowing the secret tactic. The processes associated with the participants have the lowest maximal confidentiality guest , as they must not gain any information about the bank’s secrets.

The first component c_0 of the pair $@\langle c_0, e_0 \rangle$ denotes a process’ *running confidentiality*. It denotes the highest level of secret information a process has obtained so far and thus is analogous to the pc label in imperative languages, making the type system flow-sensitive. The running confidentiality is capped by the maximal confidentiality, i.e., $c_0 \sqsubseteq c$. When defining a process, a programmer must indicate the process’ maximal confidentiality as well as the *initial* running confidentiality at which the process starts out when spawned.

An IFC type system increases the running confidentiality accordingly, whenever information of higher confidentiality is received, and disallow sends from senders with a higher or incomparable running confidentiality than the recipient. For example, the analyzer starts with the running confidentiality guest . When it receives the secret from the tactician, its running confidentiality increases to bank . After the receive, the analyzer can still send the message ask to a surveyor as the maximal confidentiality of the surveyor is bank . However, as soon as the surveyor receives this message from the analyzer, its running confidentiality increases to bank , which prevents it from sending messages to participants, whose maximal confidentiality is guest , because $\text{bank} \not\sqsubseteq \text{guest}$.

To address this limitation of existing IFC session type systems, we develop *regrading policies*. A regrading policy is polymorphic in a level f of the security lattice and certifies that, when regrading the running confidentiality to f , any secrets of confidentiality $d_s \not\sqsubseteq f$ learned so far will not affect future communications of confidentiality at most f after regrading.

To convey the regrading policy that a process must obey, we introduce *integrity annotations*, amounting to the second components in the pairs $@\langle c_0, e_0 \rangle$ and $\langle c, e \rangle$. We refer to e_0 as the *running integrity* of the process and to e as the *minimal integrity* of the process. The running integrity specifies what level a process is allowed to regrade to and is capped by the minimal integrity, i.e., $e_0 \sqsubseteq e$. For example, the surveyor process S runs at $@\langle bank, guest \rangle$ after having received the request from the analyzer, where the running integrity $guest$ licenses it to drop its running confidentiality as low as $guest$ upon tail-calling, but forces it to obey that policy too. The minimal integrity e of a process is naturally capped by its maximal confidentiality c , i.e., $e \sqsubseteq c$, because a process cannot learn (and thus drop) more secrets than it is licensed to. As a result, a process with maximal confidentiality and minimal integrity $\langle c, c \rangle$ effectively amounts to a non-regrading process.

We draw both integrity and confidentiality levels from the same security lattice, but interpret integrity levels *dually*, as usual: the lower a level in the lattice, the higher its integrity². For regrading this means that the lower the level a process regrades to, the stricter the process' policy becomes. The **SINTEGRITY** type system thus increases the running integrity of a process upon receiving from a process with a higher minimal integrity and disallows sends from a process of a higher or incomparable running integrity than the minimal integrity of the recipient (see § 5).

The process definitions in Fig. 2 only use concrete levels from the security lattice for confidentiality and integrity annotations. To increase code reusability, **SINTEGRITY** supports *security-polymorphic* process definitions. Such definitions range over security variables for confidentiality and integrity levels and may state constraints on these variables. The constraints must be satisfied upon spawning, which is checked by the **SINTEGRITY** type checker using a security theory. § 5 expands on security-polymorphic process definitions.

3.2 The need for regrading policies

While a regrading policy licenses regrading, it also imposes restrictions on a process' communication patterns to guarantee noninterference. To distill these restrictions, we next explore insecure implementations of the analyzer-surveyor example from § 2 that do not satisfy PSNI.

3.2.1 Hasty analyzer – optimization may introduce a timing attack

In the red tactic, the decision of the analyzer does not depend on the result provided by the second surveyor. Hence, one may be tempted to optimize the analyzer implementation by refraining from asking the opinion of the second surveyor in the branch corresponding to the red tactic (see A_H in Fig. 3). As appealing as this optimization seems, it leads to a leak. An attacker of confidentiality level $guest$ can simultaneously observe the sequence of messages transmitted along channels u_1 and u_2 of confidentiality $guest$, which connect the participants to the surveyors, and thus, can deduce which secret tactic was chosen: in case of the green tactic, the sequence of messages along u_1 and u_2 has the recurrence $u_1.ask; u_1.(yes/no); u_2.ask; u_2.(yes/no)$,

² We adopt the following convention to avoid any confusion: we use “running integrity”, “minimal integrity”, and “integrity level” for elements in the security lattice, and otherwise just “integrity”. Thus, when the integrity level in the lattice increases, the integrity decreases.

```

 $w_1:\text{vote}(\text{bank}, \text{guest}), w_2:\text{vote}(\text{bank}, \text{guest}), x:\text{tactic}(\text{bank}, \text{guest}) \vdash A_H :: z:\text{result}(\text{bank}, \text{bank})$ 
 $z \leftarrow A_H \leftarrow w_1, w_2, x =$ 
 $\text{case } x \text{ (green} \Rightarrow w_1.\text{ask}; \text{case } w_1 \text{ (yes} \Rightarrow w_2.\text{ask}; \text{case } w_2 \text{ (yes} \Rightarrow z.\text{buy}; (z \leftarrow A_H \leftarrow w_1, w_2, x)$ 
 $| \text{no} \Rightarrow z.\text{buy}; (z \leftarrow A_H \leftarrow w_1, w_2, x))$ 
 $| \text{no} \Rightarrow w_2.\text{ask}; \text{case } w_2 \text{ (yes} \Rightarrow z.\text{buy}; (z \leftarrow A_H \leftarrow w_1, w_2, x)$ 
 $| \text{no} \Rightarrow z.\text{sell}; (z \leftarrow A_H \leftarrow w_1, w_2, x))$ 
 $| \text{red} \Rightarrow w_1.\text{ask}; \text{case } w_1 \text{ (yes} \Rightarrow z.\text{buy}; (z \leftarrow A_H \leftarrow w_1, w_2, x)$ 
 $| \text{no} \Rightarrow z.\text{sell}; (z \leftarrow A_H \leftarrow w_1, w_2, x))) @\langle \text{guest}, \text{guest} \rangle$ 

 $w_1:\text{vote}(\text{bank}, \text{guest}), w_2:\text{vote}(\text{bank}, \text{guest}), x:\text{tactic}(\text{bank}, \text{guest}) \vdash A_R :: z:\text{result}(\text{bank}, \text{bank})$ 
 $z \leftarrow A_R \leftarrow w_1, w_2, x =$ 
 $\text{case } x \text{ (green} \Rightarrow w_1.\text{ask}; \text{case } w_1 \text{ (yes} \Rightarrow w_2.\text{ask}; \text{case } w_2 \text{ (yes} \Rightarrow z.\text{buy}; (z \leftarrow A_R \leftarrow w_1, w_2, x)$ 
 $| \text{no} \Rightarrow z.\text{buy}; (z \leftarrow A_R \leftarrow w_1, w_2, x))$ 
 $| \text{no} \Rightarrow w_2.\text{ask}; \text{case } w_2 \text{ (yes} \Rightarrow z.\text{buy}; (z \leftarrow A_R \leftarrow w_1, w_2, x)$ 
 $| \text{no} \Rightarrow z.\text{sell}; (z \leftarrow A_R \leftarrow w_1, w_2, x))$ 
 $| \text{red} \Rightarrow w_1.\text{ask}; w_2.\text{ask}; \text{case } w_1 \text{ (yes} \Rightarrow \text{case } w_2 \text{ (yes} \Rightarrow z.\text{buy}; (z \leftarrow A_R \leftarrow w_1, w_2, x)$ 
 $| \text{no} \Rightarrow z.\text{buy}; (z \leftarrow A_R \leftarrow w_1, w_2, x))$ 
 $| \text{no} \Rightarrow \text{case } w_2 \text{ (yes} \Rightarrow z.\text{sell}; (z \leftarrow A_R \leftarrow w_1, w_2, x)$ 
 $| \text{no} \Rightarrow z.\text{sell}; (z \leftarrow A_R \leftarrow w_1, w_2, x)))$ 
 $@\langle \text{guest}, \text{guest} \rangle$ 

```

 **Figure 3** Insecure hasty analyzer A_H and reckless analyzer A_R , rejected by SINTEGRITY.

whereas it has the recurrence $u_1.\text{ask}; u_1.(yes/no)$ for the red tactic. Observing, for example, the sequence $u_1.\text{ask}; u_1.(yes/no); u_2.\text{ask}; u_2.(yes/no); u_1.\text{ask}; u_1.(yes/no)$, the attacker can deduce that the first tactic used was green and the second one was red. These leaks constitute *timing attacks* because the attacker cannot deduce the secret by only looking at a single channel, but needs to observe the relative timing of messages passed along two or more channels.

3.2.2 Reckless analyzer – be careful with synchronization

The previous example shows that a send along a channel, present in one branch, but omitted from another, may lead to a leak. One may naively suspect that these leaks only involve sends. The analyzer version A_R in Fig. 3 showcases the opposite: mismatched receives are at least as dangerous as mismatched sends. In the original implementation (Fig. 2), the analyzer synchronizes the communications of surveyors and participants across branches, ensuring, in particular, that the second participant always casts their vote after the first. The reckless analyzer A_R breaks this synchronization in the red branch by swapping the order of $\text{case } w_1$ and $w_2.\text{ask}$. This minimal change allows the two surveyors to run concurrently when the tactic is red and produce the sequence of messages $u_2.\text{ask}; u_2.(yes/no); u_1.\text{ask}; u_1.(yes/no)$ along channels u_1 and u_2 , a sequence that is impossible to produce in the green tactic (recall that receives are blocking, but sends are not). Both A_H and A_R leak the secret with a timing attack, i.e., the simultaneous observation of the relative order of sends along several channels.

There is a subtle connection between timing attacks and leaks due to the non-reactivity of a process. For instance, let us assume that the second participant loops internally and never casts its vote. The attacker can then deduce the secret tactic in the hasty implementation of the analyzer by only observing the communications of the first participant along u_1 : the sequence $u_1.\text{ask}; u_1.(yes/no); u_1.\text{ask}; u_1.(yes/no)$ indicates that the prior tactic was red. A similar scenario holds for the reckless analyzer when the first participant is non-reactive.

3.3 Regrading policies in a nutshell

Our model allows the running confidentiality of a process to be dropped as low as its running integrity. Performing such a venturous act, needs a corresponding safety net in place: a regrading policy that is polymorphic in the running integrity to preserve noninterference. The examples in § 3.2 suggest that a regrading policy must enforce the following properties:

1. The continuation of a process after regrading must not depend on any secret higher than or incomparable to its running integrity. That is, when branching on a secret \mathbf{d}_s , the same process must be spawned for the recursive call in every branch, if that process regrades to a level \mathbf{e}_0 such that $\mathbf{d}_s \not\leq \mathbf{e}_0$.
2. Whether a process reaches its regrading point or not must not depend on any secret higher than or incomparable to its running integrity.

The latter property is violated in both analyzer implementations of Fig. 3, amounting to a leak. In the hasty implementation A_H , the second surveyor only gets to the regrading point if the secret tactic is green. In the reckless implementation A_R , if the secret tactic is green, the second surveyor gets to the regrading point only if the first participant casts their vote, whereas if the secret is red, there is no such chaining.

The above properties capture semantically what conditions secure processes that employ regrading must meet to observe PSNI. In § 5.2 we develop static checks that, when satisfied by a process, ensure that the process also meets these semantic conditions. We refer to those checks as *synchronization pattern* checks, and they are enforced by the **SINTEGRITY** type checker. The pattern checks are of the form $\Psi \models P \sim_{(d,f)} Q$ and synchronize P and Q in terms of their communication actions: if P outputs along channel x , so must Q , and if P inputs along channel x , so must Q , and vice versa. The pattern checks are invoked pairwise for every two branches, P_i and P_j , in a **case** statement, requiring that $\Psi \models P_i \sim_{(d,f)} P_j$. The check is conditioned on the running confidentiality d and running integrity f at the branching point.

An important feature of our regrading policies is that they are *compositional*. We take advantage of the fact that intuitionism imposes a rooted tree structure on process configurations and require that a configuration aligns with the security lattice: for every child process and parent process with maximal confidentiality and minimal integrity $\langle \mathbf{c}, \mathbf{e} \rangle$ and $\langle \mathbf{c}', \mathbf{e}' \rangle$, resp., it must hold that $\langle \mathbf{c}, \mathbf{e} \rangle \sqsubseteq \langle \mathbf{c}', \mathbf{e}' \rangle$, ensuring that a child process can learn at most as much as its parent and has at least an as stringent regrading policy as its parent. We design our type system to preserve this property as an invariant.

4 Blueprint for Formal Development

Before delving into the formal development, we review the statics and dynamics of a vanilla intuitionistic session type system and give a roadmap for the upcoming technical sections. We use the intuitionistic session type system introduced by Toninho et al. [9, 41] as our vanilla intuitionistic session type system. **SINTEGRITY** enhances such a vanilla session type system with confidentiality and integrity annotations to establish PSNI. **SINTEGRITY** adopts the former from existing intuitionistic IFC session type systems [7, 20]. The integrity annotations as well as the synchronization patterns are contributions unique to **SINTEGRITY**. The addition requires us to define the relationships between all these levels, expressed as invariants, and the development of synchronization patterns. Similar to the system in [7] our language supports general recursion and allows processes to be polymorphic in confidentiality levels. **SINTEGRITY** extends label polymorphism to also accommodate integrity levels.

4.1 Vanilla intuitionistic session types – statics

Process terms and session types are built by the grammar in § 2 and Table 1. The process typing judgment is of the form $\Delta \vdash_{\Sigma} P :: x:A$, to be read as: “*Process P provides a session of type A along channel x, given the typing of sessions offered along channels in Δ*. Δ is a linear typing context consisting of the channels connecting P to its children, and x connects P to its parent. The global signature Σ includes recursive type definitions and process definitions.

4.1.1 Process term typing

Fig. 4 lists the process term typing rules. The parts in red are specific to SINTEGRITY and can be ignored for now; we discuss them in § 5. As is usual in intuitionistic linear session type languages, the rules are given in a sequent calculus. When read from bottom to top, the rules closely follow the behavior described in Table 1: right rules describe a type from the point of view of a provider, and left rules from the point of view of a client. For example, rule \oplus_{R_1} describes the behavior of the process that provides a channel with the protocol $\oplus\{\ell:A_\ell\}_{\ell \in L}$: it chooses a label $k \in L$ and sends it to the client along channel x , and then continues by checking process P providing A_k in the premise. Note that the typing rules \oplus_{R_1} and \oplus_{R_2} are identical, ignoring the security annotations. Rule FWD ensures that the type of the two channels involved in forwarding is the same. Rule SPAWN spawns a new child process X along the fresh channel x ; it first checks that X is defined in the signature (first premise) and thus is well-typed and then continues with type-checking the continuation Q (last premise).

4.1.2 Signature checking

To support general recursive types, we employ a global signature Σ comprised of all process definitions. Each process definition is typed individually, assuming that the other processes in the signature are well-typed. The signature also comprises recursive type definitions. When typing a process with a recursive protocol, the signature is consulted to unfold the definition.

For example, the signature for the bank survey example in § 2 consists of the definitions for processes A , S , and S' as shown in Fig. 1 and the definition of recursive types as shown in Fig. 1(d). In our formal development, we use a more concise syntax for process definitions than what is shown in Fig. 1. In particular, we write them in the form of $\Delta \vdash X = P::(z:A)$. For instance, the concise version of the process definition for process S in Fig. 1, ignoring its security annotations, is $u:\text{vote} \vdash S = \text{case } w \ (\text{ask} \Rightarrow (w \leftarrow S' \leftarrow u)) :: w:\text{vote}$.

Type checking starts with typing the signature by the rules listed in Fig. 5; again, ignore the parts in red for now, as they will be discussed later in § 5. The rules are in a sequent calculus and should be read from bottom to top. Rule Σ_3 ensures that each process definition in the signature is well-typed. It invokes the process term typing judgment for a process definition relative to the entire global signature Σ (fifth premise) and continues with checking the rest of the signature (sixth premise). Rule Σ_2 ensures that all recursive types in the signature are well-formed via its first premise, the judgment $\Vdash_\Sigma A \text{ wfmd}$. This judgment denotes a well-formed session type definition, which, if recursive, must be *equi-recursive* [19] and *contractive*. Equi-recursiveness ensures that types are related up to their unfolding without requiring explicit (un)fold messages (see rules TVAR_R and TVAR_L). Contractiveness demands an exchange before recurring.

4.2 Vanilla intuitionistic session types – dynamics

At runtime, process definitions result in a configuration of processes structured as a forest of rooted trees. The nodes in the forest represent runtime processes and messages, denoted as $\text{proc}(y_\alpha; P)$ and $\text{msg}(M)$, resp. We use metavariables \mathcal{C} and \mathcal{D} to refer to a configuration and formally define it as a set of runtime processes and messages (the nodes in the tree). The connection between the nodes will be inferred through configuration typing. In $\text{proc}(y_\alpha, P)$, the metavariable y_α represents the process’ offering channel, and P represents the process’ source code (where free variables have been substituted by channels). Runtime messages $\text{msg}(M)$ are a special form of processes created to model asynchronous communication: we

implement asynchronous sends by spawning off the message $\mathbf{msg}(M)$ that carries the sent message M . A sent message M can be of the form $x.k$, $\mathbf{send} y x$, or $\mathbf{close} x$, corresponding to label output, channel output, and a termination message, resp.

Runtime channels y_α are annotated with a generation subscript α , which distinguishes them from channel variables y used in the statics. Using generation subscripts, we can ensure that both the sender and receiver agree on a new name for the continuation channel without explicitly passing the name in a message. We will see an example of using generation subscripts in the next paragraph.

4.2.1 Asynchronous dynamics

We chose an asynchronous semantics for SINTEGRITY because it weakens the attacker model, allowing a more permissive IFC enforcement, and is also a sensible model for practical purposes. The dynamics is given in Fig. 8 in terms of multiset rewriting rules [14] (again, for now the parts in red can be ignored). Multiset rewriting rules express the dynamics as state transitions between configurations and are *local* in that they only mention the parts of a configuration they rewrite.

For example, in case of $\otimes_{\mathbf{snd}}$, the provider $\mathbf{proc}(y_\alpha, \mathbf{send} x_\beta y_\alpha; P)$ spawns off the message process $\mathbf{msg}(\mathbf{send} x_\beta y_\alpha)$, indicating that the channel x_β is sent over channel y_α . Since sends are non-blocking, the provider steps to its continuation $\mathbf{proc}(y_{\alpha+1}, ([y_{\alpha+1}/y_\alpha]P))$, allocating a new generation $\alpha+1$ of the carrier channel y_α . In $\otimes_{\mathbf{recv}}$, upon receipt of the message, the receiving client process $\mathbf{proc}(y_\alpha, w \leftarrow \mathbf{recv} y_\alpha; P)$ will increment the generation of the carrier channel in its continuation. The scenario is similar for $\oplus_{\mathbf{snd}}$ and $\oplus_{\mathbf{recv}}$, but the sent message is a label in this case, and similar for $\neg o_{\mathbf{snd}}$, $\neg o_{\mathbf{recv}}$ and $\&_{\mathbf{snd}}$, $\&_{\mathbf{recv}}$, except that in these cases the sender is the client and the receiver the provider. In the rules for the termination protocol, i.e., $1_{\mathbf{snd}}$ and $1_{\mathbf{recv}}$, there is no continuation channel. Rule SPAWN creates a process offering along a fresh runtime channel x_0 by looking up the definition of the spawnee in the signature.

The dynamics for the forwarding process $\mathbf{proc}(y_\alpha, y_\alpha \leftarrow x_\beta)$ is often described as fusing the two channels, y_α and x_β . We, however, represent forward as syntactic sugar by including forwarder processes defined by structural induction on the type of the channels involved in the forward, amounting to an identity expansion. The reader may see the TR for the details.

4.2.2 Configuration typing

The configuration typing judgment is of the form $\Delta \Vdash_{\Sigma} \mathcal{C} :: \Delta'$ indicating that the configuration \mathcal{C} provides sessions along the channels in Δ' , using sessions provided along channels in Δ . Δ and Δ' are both linear contexts, consisting of actual runtime channels of the form $y_\alpha:B$. We often use the term *open configurations* to emphasize that our configurations may have external free channels in both Δ and Δ' to communicate with the environment. This is in contrast to restricting Δ to be an empty context, which means the configuration only has external free channels to communicate with a client.

Fig. 7 shows the typing rules, enforcing that the configuration is structured as a forest and the source code of each node is well-typed. For brevity, Fig. 7 omits a channel's generation as well as Σ , which is fixed. The **emp** rule types an empty forest. The **comp** rule types each tree in the forest. The **proc** rule and the **msg** rule check the well-typedness of the root node of a tree when it is a process or message, resp., using the last premises. Well-typedness of the remaining forest is checked by the eighth and fourth premise of the latter two rules, resp. The last premise of the **msg** rule calls message typing rules, which we provide in TR-Sect. 3.3.

The typing rules ensure progress and preservation, i.e., the dynamics can always step an open configuration $\Delta \Vdash \mathcal{C} :: \Delta'$ to $\Delta \Vdash \mathcal{C}' :: \Delta'$.

4.3 Roadmap for SINTEGRITY

To develop the ideas discussed in § 3 and establish PSNI, we supplement the vanilla type system with a security layer. Here, we provide a roadmap to the key parts of our development.

4.3.1 Regrading policy type system

The first step in our formal development is to enrich the process term typing judgment with security levels as $\Psi; \Delta \vdash_{\Sigma} P @ \langle c_0, e_0 \rangle :: x:A \langle c, e \rangle$. Here, Ψ denotes a security theory which includes the security lattice and polymorphic confidentiality and integrity variables. The pair $\langle c_0, e_0 \rangle$ denotes the running confidentiality and integrity of the process, aka its taint level. The pair $\langle c, e \rangle$ denotes the max confidentiality and min integrity of the process. Similarly, each channel in Δ is annotated with a pair of confidentiality and integrity levels denoting its provider's max confidentiality and min integrity.

Similarly, we use security labels to annotate configurations and configuration typing judgments. In particular, runtime processes in configuration \mathcal{C} now have the form $\text{proc}(y_{\alpha} \langle c, e \rangle, P @ \langle c', e' \rangle)$, where $\langle c, e \rangle$ is the pair of max confidentiality and min integrity of the process, and $\langle c', e' \rangle$ is the pair of its running confidentiality and integrity.

The typing rules include security constraints highlighted in red – the ones we have been ignoring in § 4.1. The purpose of these security annotations is to (i) ensure that the taint levels are propagated correctly, (ii) prevent a tainted process from sending information to a process with a lower max confidentiality/higher min integrity, (iii) ensure that a process regrades its running confidentiality only as low as its running integrity, and (iv) verify that the process indeed adheres to the policy enforced by its running integrity. The first three conditions are enforced by imposing the security constraints on the process term typing rules in Fig. 4. The last check is enforced by the synchronization pattern checks in Fig. 6.

4.3.2 PSNI via a logical relation

Our ultimate goal is to prove that well-typed SINTEGRITY processes enjoy PSNI. We prove PSNI as an equivalence up to an attacker's confidentiality level ξ using a logical relation, which then delivers a process bisimulation.

To define PSNI for an open configuration in the shape of a tree $\Psi_0; \Delta \Vdash \mathcal{D} :: u_{\alpha}:T \langle c, e \rangle$, given a global security lattice Ψ_0 fixed for an application, we consider the external free channels $y_{\beta}:B \langle c', e' \rangle \in \Delta, u_{\alpha}:T \langle c, e \rangle$ with max confidentiality $c' \sqsubseteq \xi$. We call the set of these channels that connect a configuration to its environment and that are observable to an attacker, the *confidentiality interface*.

Such an open configuration satisfies noninterference if, when composed with different high-confidentiality processes, behaves the same along the confidentiality interface. We prove that all well-typed open configurations enjoy PSNI by designing a logical relation and showing that (i) all well-typed configurations are self-related (fundamental theorem, Thm. 1) and (ii) any two related configurations are bisimilar (adequacy theorem, Thm. 3).

To prove these results, our logical relation needs to consider some free channels in $\Delta, u_{\alpha}:T \langle c, e \rangle$ that are not directly observable in terms of their confidentiality but can have an observable effect due to their integrity. We thus define a superset of the confidentiality interface that additionally contains channels $y_{\beta}:B \langle c', e' \rangle \in \Delta, u_{\alpha}:T \langle c, e \rangle$ with min integrity $e' \sqsubseteq \xi$. We call this interface the *integrity interface*.

5 Regrading policy type system

This section formalizes SINTEGRITY’s type system with synchronization patterns and asynchronous dynamics. SINTEGRITY supports security-polymorphic process definitions, an example of which is discussed in §5.4.

5.1 Process term typing

Let us recall the process term typing judgment from §4.3:

$$\Psi; \Delta \vdash_{\Sigma} P @ \langle c_0, e_0 \rangle :: x : A \langle c, e \rangle.$$

We read it as: “*Process P, with maximal confidentiality and minimal integrity $\langle c, e \rangle$ and running confidentiality and integrity $\langle c_0, e_0 \rangle$, provides a session of type A along channel x, given the typing of sessions offered along channels in Δ and given a security theory Ψ .*” Δ is a linear typing context with the grammar $\Delta ::= \cdot \mid x : A \langle c, e \rangle, \Delta$. A security theory Ψ is used for type checking security-polymorphic process definitions. It consists of the global security lattice Ψ_0 which is fixed for an application, security variables ψ , and constraints on the variables (see §5.4 and Sect. 2 in the TR).

We impose the following properties on the typing judgment, as discussed in detail in §3. These properties are maintained by typing as invariants. When reading them, note that “high integrity” and “low confidentiality” both mean a “lower level” in the security lattice.

- (a) $\forall y : B \langle d, f \rangle \in \Delta. \Psi \Vdash d \sqsubseteq c, \Psi \Vdash f \sqsubseteq e$: ensuring that a child process can learn at most as much as its parent and has at least an as stringent regrading policy as its parent.
- (b) $\Psi \Vdash c_0 \sqsubseteq c$ and $\Psi \Vdash e_0 \sqsubseteq e$: ensuring that a process knows at most as much as it is licensed to and adheres to at least an as stringent regrading policy as it promises.
- (c) $\Psi \Vdash e_0 \sqsubseteq c_0$ and $\Psi \Vdash e \sqsubseteq c$: ensuring that a process cannot drop more secrets than it knows and is licensed to learn, resp.

Moreover, the typing rules for input and output have to conform to the following schema to make sure that the running confidentiality and running integrity correctly reflect the taint level and that a tainted process does not leak information via a send:

- (1) **after** receiving a message, the running confidentiality and running integrity of the receiving process must be increased to **at least** the maximal confidentiality and minimal integrity of the sending process, and
- (2) **Before** sending a message, the running confidentiality and running integrity of the sending process must be **at most** the maximal confidentiality and minimal integrity of the receiving process.

Conforming to this schema leads to the premises of the form $\Psi \Vdash \langle d_1, f_1 \rangle = \langle c, e \rangle \sqcup \langle d_0, f_0 \rangle$ and $\Psi \Vdash \langle d_0, f_0 \rangle \sqsubseteq \langle c, e \rangle$ to meet condition (1) and (2), resp., above. The judgments are defined formally in Sect. 2 in the TR.

It is time to consider the red security annotations of the typing rules in Fig. 4. We explain how the rules satisfy conditions (1) and (2) above:

- \oplus : There are two versions of the right rule for \oplus . Both versions establish condition (2) on sends without extra premises by the invariant (b). The difference between the two versions lies in whether $\Psi \Vdash c = e$ is derivable or not derivable ($\Psi \nVdash c = e$). If $\Psi \Vdash c = e$ is derivable, then rule \oplus_{R_1} applies; if it is not, rule \oplus_{R_2} applies. In the former case, the client of x , on the receiving side, adjusts its running integrity to at least $e=c$ upon receiving the sent message, and thus, it cannot regrade to a lower (or unrelated) level than c . In the latter case, the min integrity e of the process is strictly lower than its max confidentiality c . This means that the client of x might, in fact, continue to have

$$\begin{array}{c}
 \frac{\Psi \Vdash c = e \quad k \in L \quad \Psi; \Delta \vdash_{\Sigma} P @ \langle c_0, e_0 \rangle :: x:A_k \langle c, e \rangle}{\Psi; \Delta \vdash_{\Sigma} (x^{\langle c, e \rangle}.k; P) @ \langle c_0, e_0 \rangle :: x: \oplus \{\ell:A_{\ell}\}_{\ell \in L} \langle c, e \rangle} \oplus_{R_1} \\
 \frac{\Psi \nVdash c = e \quad \forall i, j \in L. A_i = A_j \quad k \in L \quad \Psi; \Delta \vdash_{\Sigma} P @ \langle c_0, e_0 \rangle :: x:A_k \langle c, e \rangle}{\Psi; \Delta \vdash_{\Sigma} (x^{\langle c, e \rangle}.k; P) @ \langle c_0, e_0 \rangle :: x: \oplus \{\ell:A_{\ell}\}_{\ell \in L} \langle c, e \rangle} \oplus_{R_2} \\
 \frac{\Psi \Vdash \langle d_1, f_1 \rangle = \langle c, e \rangle \sqcup \langle d_0, f_0 \rangle \quad \forall k \in L \quad \Psi; \Delta, x:A_k \langle c, e \rangle \vdash_{\Sigma} Q_k @ \langle d_1, f_1 \rangle :: z:C \langle d, f \rangle \quad \forall i, j \in L. \Psi \models Q_i \sim_{\langle d_1, f_1 \rangle} Q_j}{\Psi; \Delta, x: \oplus \{\ell:A_{\ell}\}_{\ell \in L} \langle c, e \rangle \vdash_{\Sigma} (\text{case } x^{\langle c, e \rangle} (\ell \Rightarrow Q_{\ell})_{\ell \in L}) @ \langle d_0, f_0 \rangle :: z:C \langle d, f \rangle} \oplus_L \\
 \frac{\forall k \in L \quad \Psi; \Delta \vdash_{\Sigma} P_k @ \langle c, e \rangle :: x:A_k \langle c, e \rangle \quad \forall i, j \in L. \Psi \models P_i \sim_{\langle c, e \rangle} P_j}{\Psi; \Delta \vdash_{\Sigma} (\text{case } x^{\langle c, e \rangle} (\ell \Rightarrow P_{\ell})_{\ell \in L}) @ \langle c_0, e_0 \rangle :: x: \& \{\ell:A_{\ell}\}_{\ell \in L} \langle c, e \rangle} \&_R \\
 \frac{\Psi \Vdash c = e \quad \Psi \Vdash \langle d_0, f_0 \rangle \sqsubseteq \langle c, e \rangle \quad k \in L \quad \Psi; \Delta, x:A_k \langle c, e \rangle \vdash_{\Sigma} Q @ \langle d_0, f_0 \rangle :: z:C \langle d, f \rangle}{\Psi; \Delta, x: \& \{\ell:A_{\ell}\}_{\ell \in L} \langle c, e \rangle \vdash_{\Sigma} (x^{\langle c, e \rangle}.k; Q) @ \langle d_0, f_0 \rangle :: z:C \langle d, f \rangle} \&_{L_1} \\
 \frac{\Psi \nVdash c = e \quad \forall i, j \in L. A_i = A_j \quad \Psi \Vdash \langle d_0, f_0 \rangle \sqsubseteq \langle c, e \rangle \quad k \in L \quad \Psi; \Delta, x:A_k \langle c, e \rangle \vdash_{\Sigma} Q @ \langle d_0, f_0 \rangle :: z:C \langle d, f \rangle}{\Psi; \Delta, x: \& \{\ell:A_{\ell}\}_{\ell \in L} \langle c, e \rangle \vdash_{\Sigma} (x^{\langle c, e \rangle}.k; Q) @ \langle d_0, f_0 \rangle :: z:C \langle d, f \rangle} \&_{L_2} \\
 \frac{\Psi; \Delta \vdash_{\Sigma} P @ \langle c_0, e_0 \rangle :: x:B \langle c, e \rangle}{\Psi; \Delta, y:A \langle d, f \rangle \vdash_{\Sigma} (\text{send } y x^{\langle c, e \rangle}; P) @ \langle c_0, e_0 \rangle :: x:A \otimes B \langle c, e \rangle} \otimes_R \\
 \frac{\Psi \Vdash \langle d_1, f_1 \rangle = \langle c, e \rangle \sqcup \langle d_0, f_0 \rangle \quad \Psi; \Delta, x:B \langle c, e \rangle, y:A \langle c, e \rangle \vdash_{\Sigma} Q @ \langle d_1, f_1 \rangle :: z:C \langle d, f \rangle}{\Psi; \Delta, x:A \otimes B \langle c, e \rangle \vdash_{\Sigma} (y^{\langle c, e \rangle} \leftarrow \text{recv } x^{\langle c, e \rangle}; Q_y @ \langle d_0, f_0 \rangle :: z:C \langle d, f \rangle)} \otimes_L \\
 \frac{\Psi; \Delta, y:A \langle c, e \rangle \vdash_{\Sigma} P @ \langle c, e \rangle :: x:B \langle c, e \rangle}{\Psi; \Delta \vdash_{\Sigma} (y^{\langle c, e \rangle} \leftarrow \text{recv } x^{\langle c, e \rangle}; P_y @ \langle c_0, e_0 \rangle :: x:A \multimap B \langle c, e \rangle)} \multimap_R \\
 \frac{\Psi \Vdash \langle d_0, f_0 \rangle \sqsubseteq \langle c, e \rangle \quad \Psi; \Delta, x:B \langle c, e \rangle \vdash_{\Sigma} Q @ \langle d_0, f_0 \rangle :: z:C \langle d, f \rangle}{\Psi; \Delta, x:A \multimap B \langle c, e \rangle, y:A \langle c, e \rangle \vdash_{\Sigma} (\text{send } y x^{\langle c, e \rangle}; Q) @ \langle d_0, f_0 \rangle :: z:C \langle d, f \rangle} \multimap_L \\
 \frac{\Psi \Vdash \langle c_1, e_1 \rangle = \langle c_2, e_2 \rangle}{\Psi; y:A \langle c_1, e_1 \rangle \vdash_{\Sigma} (x^{\langle c_2, e_2 \rangle} \leftarrow y^{\langle c_1, e_1 \rangle}) @ \langle c_0, e_0 \rangle :: x:A \langle c_2, e_2 \rangle} \text{FWD} \\
 \frac{\Psi' ; \Delta'_1 \vdash_{\Sigma} X = P @ \langle \psi_0, \omega_0 \rangle :: x:A \langle \psi, \omega \rangle \in \Sigma \quad \Psi \Vdash \gamma : \Psi' \quad \hat{\gamma}(\Delta'_1) = \Delta_1 \quad \Psi \Vdash \langle \hat{\gamma}(\psi), \hat{\gamma}(\omega) \rangle \sqsubseteq \langle d, f \rangle \quad \Psi \Vdash f_0 \sqsubseteq \hat{\gamma}(\psi_0) \quad \Psi \Vdash f_0 \sqsubseteq \hat{\gamma}(\omega_0) \quad \Psi; \Delta_2, x:A \langle \hat{\gamma}(\psi), \hat{\gamma}(\omega) \rangle \vdash_{\Sigma} Q @ \langle d_0, f_0 \rangle :: z:C \langle d, f \rangle}{\Psi; \Delta_1, \Delta_2 \vdash_{\Sigma} ((x^{\langle \hat{\gamma}(\psi), \hat{\gamma}(\omega) \rangle} \leftarrow X[\gamma] \leftarrow \Delta_1) @ \langle \hat{\gamma}(\psi_0), \hat{\gamma}(\omega_0) \rangle; Q_x) @ \langle d_0, f_0 \rangle :: z:C \langle d, f \rangle} \text{SPAWN} \\
 \frac{}{\Psi; \cdot \vdash_{\Sigma} (\text{close } x^{\langle c, e \rangle}) @ \langle c_0, e_0 \rangle :: x:1 \langle c, e \rangle} \mathbf{1}_R \\
 \frac{\Psi \Vdash \langle d_1, f_1 \rangle = \langle c, e \rangle \sqcup \langle d_0, f_0 \rangle \quad \Psi; \Delta \vdash_{\Sigma} Q @ \langle d_1, f_1 \rangle :: z:C \langle d, f \rangle}{\Psi; \Delta, x:1 \langle c, e \rangle \vdash_{\Sigma} (\text{wait } x^{\langle c, e \rangle}; Q) @ \langle d_0, f_0 \rangle :: z:C \langle d, f \rangle} \mathbf{1}_L
 \end{array}$$

Figure 4 Process term typing rules of SINTEGRITY.

its running integrity as low as $e \sqsubset c$ and, at some point in the future, drop its running confidentiality to e and start sending to channels with lower (or unrelated) confidentiality than c . The additional premise $\forall i, j \in L. A_i = A_j$ in \oplus_{R_2} prevents potential leaks through different continuation protocols at that future point, i.e., it ensures that the client's future communications with channels of lower confidentiality level than c do not depend on the continuation protocol chosen now.

$$\begin{array}{c}
 \text{TVAR}_R \\
 \frac{Y = A \in \Sigma \quad \Psi; \Delta \vdash_{\Sigma} P @ \langle c_0, e_0 \rangle :: x:A \langle c, e \rangle}{\Psi; \Delta \vdash_{\Sigma} P @ \langle c_0, e_0 \rangle :: x:Y \langle c, e \rangle}
 \end{array}$$

$$\begin{array}{c}
 \text{TVAR}_L \\
 \frac{Y = A \in \Sigma \quad \Psi; \Delta, x:A \langle c, e \rangle \vdash_{\Sigma} Q @ \langle d_0, f_0 \rangle :: z:C \langle d, f \rangle}{\Psi; \Delta, x:Y \langle c, e \rangle \vdash_{\Sigma} Q @ \langle d_0, f_0 \rangle :: z:C \langle d, f \rangle}
 \end{array}
 \quad \frac{}{\vdash_{\Sigma; \Psi_0} (\cdot) \text{ sig}} \Sigma_1$$

$$\frac{\vdash_{\Sigma} A \text{ wfmd} \quad \vdash_{\Sigma; \Psi_0} \Sigma' \text{ sig}}{\vdash_{\Sigma; \Psi_0} Y = A, \Sigma' \text{ sig}} \Sigma_2$$

$$\frac{\forall i \in \{1 \dots n\}. \Psi \Vdash \langle \psi_i, \omega_i \rangle \sqsubseteq \langle \psi, \omega \rangle, \Psi \Vdash \omega_i \sqsubseteq \psi_i \quad \Psi \Vdash \langle \psi_0, \omega_0 \rangle \sqsubseteq \langle \psi, \omega \rangle \quad \Psi \Vdash \omega_0 \sqsubseteq \psi_0 \quad \Psi \Vdash \omega \sqsubseteq \psi}{\Psi; y_1:B_1 \langle \psi_1, \omega_1 \rangle, \dots, y_n:B_n \langle \psi_n, \omega_n \rangle \vdash_{\Sigma} P @ \langle \psi_0, \omega_0 \rangle :: x:A \langle \psi, \omega \rangle \quad \vdash_{\Sigma; \Psi_0} \Sigma' \text{ sig}} \Sigma_3$$

$$\frac{}{\vdash_{\Sigma; \Psi_0} \Psi; y_1:B_1 \langle \psi_1, \omega_1 \rangle, \dots, y_n:B_n \langle \psi_n, \omega_n \rangle \vdash X = P @ \langle \psi_0, \omega_0 \rangle :: x:A \langle \psi, \omega \rangle, \Sigma' \text{ sig}}$$

Figure 5 Signature checking rules of SINTEGRITY.

The first premise of rule $\oplus L$ updates the running integrity and confidentiality based on x 's security levels to enforce condition (1) for receives. Moreover, as explained in § 3.3, the third premise invokes the pattern check pairwise for every two branches conditioned on the running confidentiality d_1 and running integrity f_1 after the receive. We detail the synchronization pattern check rules later in § 5.2.

- $\&$: The left and right rules for $\&$ are dual to \oplus , except that the sends in $\&_{L_1}$ and $\&_{L_2}$ have to be guarded by their second and third premises, resp., to ensure condition (2) on sends. In $\&_R$, the updated running confidentiality and running integrity is equal to the max confidentiality and max integrity by invariant (b).
- $\otimes, \multimap, 1$: The rules for the rest of the connectives use the same set of premises to ensure conditions (1) and (2). Rules \otimes_R and \multimap_L , moreover, ensure that a channel can be sent over another channel only if they have the same security levels.
- FWD: The forward rule requires that the security levels of the involved channels match.
- SPAWN: The rule relies on an order-preserving substitution $\Psi \Vdash \gamma : \Psi'$, guaranteeing that the security terms provided by the spawner comply with the order expected among those terms by the spawnee. The substitution maps the security terms in the context in the signature to the one provided by the spawner, i.e., $\hat{\gamma}(\Delta'_1) = \Delta_1$. The rule also establishes invariants (a)-(c) for the newly spawned process via the premise $\Psi \Vdash \langle \hat{\gamma}(\psi), \hat{\gamma}(\omega) \rangle \sqsubseteq \langle d, f \rangle$. The running confidentiality and the running integrity of the spawned process will result from applying the substitution to the corresponding levels in the signature, i.e., $\hat{\gamma}(\psi_0)$ and $\hat{\gamma}(\omega_0)$, resp. The premises $\Psi \Vdash f_0 \sqsubseteq \hat{\gamma}(\psi_0)$ and $\Psi \Vdash f_0 \sqsubseteq \hat{\gamma}(\omega_0)$ allow the newly spawned process to start its running confidentiality and integrity at least at the spawner's running integrity f_0 . This facilitates regrading to f_0 in case of a tail call. Note that $\Psi \Vdash f_0 \sqsubseteq \hat{\gamma}(\omega_0)$ prevents the spawnee from employing more pattern checks than the spawner because the spawnee would otherwise be affected by the spawners negligence.

Signature checking. The syntax of process definitions in the signature is also enhanced with the security levels and is of the form $\Psi; \Delta \vdash X = P @ \langle \psi_0, \omega_0 \rangle :: (z:A \langle \psi, \omega \rangle)$. Fig. 5 lists the signature checking rules. Signature checking happens relative to a globally fixed security lattice Ψ_0 of concrete security levels. Rule Σ_3 initiates type-checking of a process definition via its fifth premise and enforces invariants (a)-(c) on the process via the first four premises.

$$\begin{array}{c}
 \text{UNSYNC}_1 \quad \text{UNSYNC}_2 \\
 \frac{\Psi \not\models d \sqsubseteq f \quad \Psi \Vdash d \sqsubseteq e \quad \Psi \models P \sim_{\langle d,f \rangle} Q}{\Psi \models \uparrow_{x^{\langle c,e \rangle}} .P \sim_{\langle d,f \rangle} Q} \quad \frac{\Psi \not\models d \sqsubseteq f \quad \Psi \Vdash d \sqsubseteq e \quad \Psi \models P \sim_{\langle d,f \rangle} Q}{\Psi \models P \sim_{\langle d,f \rangle} \uparrow_{x^{\langle c,e \rangle}} .Q} \\
 \frac{\Psi \Vdash d \sqsubseteq f}{\Psi \models P \sim_{\langle d,f \rangle} Q} \text{ UNSYNC}_3 \\
 \frac{\Psi \not\models d \sqsubseteq f \quad \Psi \Vdash d \sqsubseteq e_0 \quad \forall y:B\langle c',e' \rangle \in \Delta. \Psi \Vdash d \sqsubseteq e' \quad \Psi \models P \sim_{\langle d,f \rangle} Q}{\Psi \models (x^{\langle c,e \rangle} \leftarrow X[\gamma] \leftarrow \Delta) @ \langle c_0, e_0 \rangle; P_x \sim_{\langle d,f \rangle} Q} \text{ UNSYNC-SPAWN}_1 \\
 \frac{\Psi \not\models d \sqsubseteq f \quad \Psi \Vdash d \sqsubseteq e_0 \quad \forall y:B\langle c',e' \rangle \in \Delta. \Psi \Vdash d \sqsubseteq e' \quad \Psi \models P \sim_{\langle d,f \rangle} Q}{\Psi \models P \sim_{\langle d,f \rangle} (x^{\langle c,e \rangle} \leftarrow X[\gamma] \leftarrow \Delta) @ \langle c_0, e_0 \rangle; Q_x} \text{ UNSYNC-SPAWN}_2 \\
 \text{SNDLAB} \\
 \frac{\Psi \not\models d \sqsubseteq f \quad \Psi \not\models d \sqsubseteq e \quad \Psi \models P \sim_{\langle d,f \rangle} Q}{\Psi \models x^{\langle c,e \rangle}.k; P \sim_{\langle d,f \rangle} x^{\langle c,e \rangle}.\ell; Q} \\
 \text{RCVLAB} \\
 \frac{\Psi \not\models d \sqsubseteq f \quad \forall j \in I, k \in L. \Psi \models P_j \sim_{\langle d,f \sqcup e \rangle} Q_k}{\Psi \models \mathbf{case} x^{\langle c,e \rangle} (\ell \Rightarrow P_\ell)_{\ell \in I} \sim_{\langle d,f \rangle} \mathbf{case} x^{\langle c,e \rangle} (\ell \Rightarrow Q_\ell)_{\ell \in L}} \\
 \text{SNDCHN} \\
 \frac{\Psi \not\models d \sqsubseteq f \quad \Psi \not\models d \sqsubseteq e \quad \Psi \models P \sim_{\langle d,f \rangle} Q}{\Psi \models \mathbf{send} y x^{\langle c,e \rangle}; P \sim_{\langle d,f \rangle} \mathbf{send} y x^{\langle c,e \rangle}; Q} \\
 \text{RCVCHN} \\
 \frac{\Psi \not\models d \sqsubseteq f \quad \Psi \models [y/y_1]P \sim_{\langle d,f \sqcup e \rangle} [y/y_2]Q}{\Psi \models y_1 \leftarrow \mathbf{recv} x^{\langle c,e \rangle}; P_{y_1} \sim_{\langle d,f \rangle} y_2 \leftarrow \mathbf{recv} x^{\langle c,e \rangle}; Q_{y_2}} \\
 \frac{\Psi \not\models d \sqsubseteq f \quad (\Psi \not\models d \sqsubseteq e_0 \text{ or } \exists y:B\langle c',e' \rangle \in \Delta. \Psi \not\models d \sqsubseteq e') \quad \Psi \models [x/x_1]P \sim_{\langle d,f \rangle} [x/x_2]Q}{\Psi \models (x_1^{\langle c,e \rangle} \leftarrow X[\gamma] \leftarrow \Delta) @ \langle c_0, e_0 \rangle; P_{x_1} \sim_{\langle d,f \rangle} (x_2^{\langle c,e \rangle} \leftarrow X[\gamma] \leftarrow \Delta) @ \langle c_0, e_0 \rangle; Q_{x_2}} \text{ SYNC-SPAWN} \\
 \frac{\Psi \not\models d \sqsubseteq f}{\Psi \models x^{\langle c_1,e_1 \rangle} \leftarrow y^{\langle c_2,e_2 \rangle} \sim_{\langle d,f \rangle} x^{\langle c_1,e_1 \rangle} \leftarrow y^{\langle c_2,e_2 \rangle}} \text{ FWD} \\
 \frac{\Psi \not\models d \sqsubseteq f}{\Psi \models \mathbf{close} x^{\langle c,e \rangle} \sim_{\langle d,f \rangle} \mathbf{close} x^{\langle c,e \rangle}} \text{ CLOSE} \quad \frac{\Psi \not\models d \sqsubseteq f \quad \Psi \models P \sim_{\langle d,f \sqcup e \rangle} Q}{\Psi \models \mathbf{wait} x^{\langle c,e \rangle}; P \sim_{\langle d,f \rangle} \mathbf{wait} x^{\langle c,e \rangle}; Q} \text{ WAIT}
 \end{array}$$

Figure 6 Synchronization pattern checking rules of SINTEGRITY.

5.2 Synchronization patterns

To check synchronization patterns, we use the judgment $\Psi \models P \sim_{\langle d,f \rangle} Q$, defined inductively in Fig. 6. The judgment states that process terms P and Q are *synchronized* in terms of their communication pattern, meaning that if P outputs along channel x , so must Q , and that if P inputs along channel x , so must Q , and vice versa. The check is *conditioned* on the running confidentiality d and running integrity f of the recipient after branching, and is pairwise called for all branches of a **case** statement. Let us assume that right after branching, the known secret of a process (its running confidentiality) is of level d . The goal of the synchronization pattern checks is to rule out any leakage of this secret of level d via regrading. Such leakage is only possible if the process (or any process that receives this secret from it) regrades to a lower or unrelated level than the secret d . However, if $d \sqsubseteq f$, we know that this can never happen. Therefore, if $\Psi \Vdash d \sqsubseteq f$, the judgment $\Psi \models P \sim_{\langle d,f \rangle} Q$ trivially holds. This case is handled by Rule UNSYNC₃ and is a base case of the inductive definition.

The interesting case is when $\Psi \not\models d \sqsubseteq f$, meaning that the process can potentially regrade to a lower (or unrelated) level than d . In this case, the rules have to ensure that the secret d does not affect the ability of the process itself or the processes communicating with it to reach a regrading point. Furthermore, the secret d cannot affect the continuation of the process after regrading. In this case, the rules consider whether the next action in P and Q is a receive, send (except close), spawn, close, or forward:

- The receives are checked to be synchronized in P and Q by the rules RCVLAB and RCVCHAN. The pattern check is invoked inductively on the continuation, with updated running integrity ($f \sqcup e$) to take into account the receive. The confidentiality of the learned secret d , however, remains constant under inductive invocations as it has to continue preventing the leak of the original secret. The receives have to be synchronized as long as $\Psi \not\models d \sqsubseteq f$ holds, since different receives in P and Q might result in one branch reaching the regrading point and the other one not (related to non-reactiveness).
- Different sends in two branches of a process does not impact whether or not the process itself reaches a regrading point (sends are non-blocking). But, it may impact whether or not the other process, on the receiving side, reaches the regrading point based on the secret. If the carrier channel's min integrity e is high enough, the receiving process cannot regrade to a level lower (or unrelated) than d , and we do not need to synchronize the sends. The sends must only be synchronized if the carrier channel's min integrity e is not greater than or equal to the level d of the secret ($d \not\sqsubseteq e$). Rules UNSYNC₁ and UNSYNC₂ correspond to the former case where $d \sqsubseteq e$; for brevity, in these rules, we use process terms with any output prefix defined as $\uparrow_{x^{(c,e)}} .P \triangleq x^{(c,e)}.k; P \mid \text{send } y x^{(c,e)}; P$. And rules SNDLAB and SNDCHAN correspond to the latter where $d \not\sqsubseteq e$. In either case, the pattern check is invoked inductively on the continuation, with unchanged running integrity.
- Similar to the reasoning in the case of sends, if the running integrity of the spawned process and the min integrity of all its channels are high enough, there is no need to synchronize the spawns (UNSYNC-SPAWN rules). Otherwise, the two branches must spawn the same processes with the same arguments (SYNC-SPAWN).
- Rules CLOSE and FWD are the other base cases of the inductive definition. They insist that the two branches P and Q can synchronize their termination behavior.

5.3 Configuration typing and asynchronous dynamics

The configuration typing judgment is of the form $\Psi_0; \Delta \Vdash \mathcal{C} :: \Delta'$, where Ψ_0 is the security lattice and \mathcal{C} is a set of runtime processes $\text{proc}(y_\alpha(c, e), P @ (c', e'))$ and messages $\text{msg}(M)$. Fig. 7 shows the configuration typing. The security premises in the **proc** and **msg** rules enforce the invariants (a)-(c) on the process term judgment before invoking process typing.

The dynamic rules in Fig. 8 take care of updating the running confidentiality and integrity of each process after a receive. For brevity, we write $\langle p \rangle$ to refer to a pair of confidentiality and integrity labels $\langle c, e \rangle$. Rule SPAWN relies on the substitution mapping γ given by the programmer and its lifting $\hat{\gamma}$ to the process term level. It looks up the definition of process X in the signature and instantiates the security variables occurring in the process body using γ . The condition $\hat{\gamma}(\Psi') = \Psi_0$ ensures that all security variables are instantiated with a concrete security level. For brevity, we omit a channel generations as well as Σ , which is fixed.

5.4 Banking example

The following example implements a bank that authorizes transactions made by its customers and sends a copy to their bank accounts. In line with our security lattice, we assume that the bank has two customers, Alice and Bob. To authenticate themselves, a customer sends their

$\Psi_o; x:A[\langle d, e \rangle] \Vdash \cdot :: (x:A[\langle d, e \rangle])$	$\Psi_o; \Delta_0 \Vdash C :: \Delta$ $\Psi_o; \Delta'_0 \Vdash C_1 :: x:A[\langle d, e \rangle]$	emp comp
$\forall y:B[\langle d', e' \rangle] \in \Delta'_0, \Delta(\Psi_o \Vdash d' \sqsubseteq d)$ $\Psi_o \Vdash e_1 \sqsubseteq d_1$	$\Psi_o \Vdash e_1 \sqsubseteq e$ $\forall y:B[\langle d', e' \rangle] \in \Delta'_0, \Delta(\Psi_o \Vdash e' \sqsubseteq e)$ $\Psi_o \Vdash e \sqsubseteq d$	
$\forall y:B[\langle d', e' \rangle] \in \Delta'_0, \Delta(\Psi_o \Vdash e' \sqsubseteq d')$ $\Psi_o; \Delta_0 \Vdash C :: \Delta$	$\forall y:B[\langle d', e' \rangle] \in \Delta'_0, \Delta(\Psi_o \Vdash e' \sqsubseteq d')$ $\Psi_o; \Delta'_0, \Delta \vdash P @ \langle d_1, e_1 \rangle :: (x:A[\langle d, e \rangle])$	$\Psi_o; \Delta'_0, \Delta \vdash P @ \langle d_1, e_1 \rangle :: (x:A[\langle d, e \rangle])$
$\Psi_o; \Delta_0, \Delta'_0 \Vdash C \text{ proc}(x[\langle d, e \rangle], P @ \langle d_1, e_1 \rangle) :: (x:A[\langle d, e \rangle])$		proc
$\forall y:B[\langle d', e' \rangle] \in \Delta'_0, \Delta(\Psi_o \Vdash d' \sqsubseteq d)$ $\Psi_o; \Delta_0 \Vdash C :: \Delta$	$\Psi_o \Vdash e \sqsubseteq d$ $\forall y:B[\langle d', e' \rangle] \in \Delta'_0, \Delta(\Psi_o \Vdash e' \sqsubseteq d')$ $\Psi_o; \Delta'_0, \Delta \vdash M @ \langle d, e \rangle :: (x:A[\langle d, e \rangle])$	$\Psi_o; \Delta_0, \Delta'_0 \Vdash C, \text{msg}(M) :: (x:A[\langle d, e \rangle])$
		msg

Figure 7 Configuration typing rules of SINTEGRITY.

SPAWN	proc ($y_\alpha \langle p \rangle$, ($x^{(p')} \leftarrow X[\gamma] \leftarrow \Delta$)@ $\langle p_2 \rangle$; $Q @ \langle p_1 \rangle$)	$(\Psi'; \Delta' \vdash X = P @ \langle \psi_0, \omega_0 \rangle :: x : B' \langle \psi, \omega \rangle \in \Sigma)$
	$\mapsto \text{proc}(x_0 \langle p' \rangle, ([x_0, \Delta/x, \Delta'] \hat{\gamma}(P)) @ \langle p_2 \rangle) \text{ proc}(y_\alpha \langle p \rangle, ([x_0/x] Q) @ \langle p_1 \rangle)$	$(\Psi_0 \Vdash \gamma : \Psi', x_0 \text{ fresh})$
1_{snd}	proc ($y_\alpha \langle p \rangle$, (close y_α)@ $\langle p_1 \rangle$)	$\mapsto \text{msg}(\text{close } y_\alpha)$
1_{rcv}	$\text{msg}(\text{close } y_\alpha) \text{ proc}(x_\beta \langle p' \rangle, (\text{wait } y_\alpha; Q) @ \langle p_1 \rangle)$	$\mapsto \text{proc}(x_\beta \langle p' \rangle, Q @ \langle p_1 \rangle \sqcup \langle p \rangle)$
\oplus_{snd}	proc ($y_\alpha \langle p \rangle$, $y_\alpha.k$; $P @ \langle p_1 \rangle$)	$\mapsto \text{proc}(y_{\alpha+1} \langle p \rangle, ([y_{\alpha+1}/y_\alpha] P) @ \langle p_1 \rangle) \text{ msg}(y_\alpha.k)$
\oplus_{rcv}	$\text{msg}(y_\alpha.k) \text{ proc}(u_\gamma \langle p' \rangle, \text{case } y_\alpha^{(p)} ((\ell \Rightarrow P_\ell)_{\ell \in L}) @ \langle p_1 \rangle)$	$\mapsto \text{proc}(u_\gamma \langle p' \rangle, ([y_{\alpha+1}/y_\alpha] P_k) @ \langle p_1 \rangle \sqcup \langle p \rangle)$
$\&_{\text{snd}}$	proc ($y_\alpha \langle p \rangle$, $(x_\beta.k; P) @ \langle p_1 \rangle$)	$\mapsto \text{msg}(x_\beta.k) \text{ proc}(y_\alpha \langle p \rangle, ([x_{\beta+1}/x_\beta] P) @ \langle p_1 \rangle)$
$\&_{\text{rcv}}$	$\text{proc}(y_\alpha \langle p \rangle, (\text{case } y_\alpha (\ell \Rightarrow P_\ell)_{\ell \in L}) @ \langle p_1 \rangle) \text{ msg}(y_\alpha.k)$	$\mapsto \text{proc}(v_\delta \langle p \rangle, ([y_{\alpha+1}/y_\alpha] P_k) @ \langle p \rangle)$
\otimes_{snd}	proc ($y_\alpha \langle p \rangle$, (send $x_\beta y_\alpha$; P)@ $\langle p_1 \rangle$)	$\mapsto \text{proc}(y_{\alpha+1} \langle p \rangle, ([y_{\alpha+1}/y_\alpha] P) @ \langle p_1 \rangle) \text{ msg}(\text{send } x_\beta y_\alpha)$
\otimes_{rcv}	$\text{msg}(\text{send } x_\beta y_\alpha) \text{ proc}(u_\gamma \langle p' \rangle, (w \leftarrow \text{recv } y_\alpha^{(p)}; P) @ \langle p_1 \rangle)$	$\mapsto \text{proc}(u_\gamma \langle p' \rangle, ([x_\beta/w][y_{\alpha+1}/y_\alpha] P) @ \langle p_1 \rangle \sqcup \langle p \rangle)$
\multimap_{snd}	proc ($y_\alpha \langle p \rangle$, (send $x_\beta u_\gamma$; P)@ $\langle p_1 \rangle$)	$\mapsto \text{msg}(\text{send } x_\beta u_\gamma) \text{ proc}(y_\alpha \langle p \rangle, ([u_{\gamma+1}/u_\gamma] P) @ \langle p_1 \rangle)$
\multimap_{rcv}	$\text{proc}(y_\alpha \langle p \rangle, (w \leftarrow \text{recv } y_\alpha; P) @ \langle p_1 \rangle) \text{ msg}(\text{send } x_\beta y_\alpha)$	$\mapsto \text{proc}(v_\delta \langle p \rangle, ([x_\beta/w][y_{\alpha+1}/y_\alpha] P) @ \langle p \rangle)$

Figure 8 Asynchronous dynamics of SINTEGRITY.

token to the bank. The bank then verifies the token and, if the verification is successful, sends the message *succ* to the customer, otherwise the message *fail*. Moreover, if the verification is successful, the bank creates a transaction statement and sends it to another process that represents the account of the customer in the bank. Once done, the bank continues to serve the next customer by making a recursive call. We assume that the bank alternates between its two customers, Alice and Bob, by making a mutually recursive call from `BankA`, which serves Alice, to `BankB`, which serves Bob, and vice versa. At each recursive call, a bank process regrades its running confidentiality to interact with the next customer. The example showcases a characteristic feature of our type system: it accepts an implementation for a bank that interactively communicates with Alice and Bob without jeopardizing noninterference.

The following session types dictate the above protocol:

$$\begin{array}{ll} \text{customer} = \oplus\{\text{tok}_{black} : \&\{\text{succ} : \text{customer}, \text{fail} : \text{customer}\}, \\ \quad \text{tok}_{white} : \&\{\text{succ} : \text{customer}, \text{fail} : \text{customer}\}\} & \text{account} = \text{transfer} \multimap \text{account} \\ & \text{transfer} = \oplus\{\text{transaction} : 1\} \end{array}$$

Fig. 9 shows the process implementations Bank_A , Bank_B , Customer_A , and Statement_A . The latter two are the implementation of Alice's customer and statement process, resp. The implementation of Customer_A is as expected. Statement_A signals a single transfer by sending the label transaction and terminates. The implementation of corresponding processes for Bob,

```

 $\Psi; y_1:\text{customer}(\psi, \text{guest}), y_2:\text{customer}(\psi', \text{guest}), w_1:\text{account}(\psi, \psi), w_2:\text{account}(\psi', \psi')$ 
 $\vdash \text{Bank}_A :: x:1\langle \text{bank}, \text{bank} \rangle$ 
 $x \leftarrow \text{Bank}_A \leftarrow y_1, y_2, w_1, w_2 =$ 
 $\text{case } y_1 \text{ (tok}_{white} \Rightarrow y_1.\text{succ}; (u \leftarrow \text{Statement}_A[\gamma] \leftarrow \cdot); \text{send } u w_1; (x' \leftarrow \text{Bank}_B[\gamma] \leftarrow y_1, y_2, w_1, w_2); x \leftarrow x'$ 
 $| \text{tok}_{black} \Rightarrow y_1.\text{fail}; (x' \leftarrow \text{Bank}_B[\gamma] \leftarrow y_1, y_2, w_1, w_2); x \leftarrow x') @ \langle \text{guest}, \text{guest} \rangle$ 
 $\Psi; y_1:\text{customer}(\psi, \text{guest}), y_2:\text{customer}(\psi', \text{guest}), w_1:\text{account}(\psi, \psi), w_2:\text{account}(\psi', \psi')$ 
 $\vdash \text{Bank}_B :: x:1\langle \text{bank}, \text{bank} \rangle$ 
 $x \leftarrow \text{Bank}_B \leftarrow y_1, y_2, w_1, w_2 =$ 
 $\text{case } y_2 \text{ (tok}_{black} \Rightarrow y_2.\text{succ}; (u \leftarrow \text{B}[\gamma] \leftarrow \cdot); \text{send } u w_2; (x' \leftarrow \text{Bank}_A[\gamma] \leftarrow y_1, y_2, w_1, w_2); x \leftarrow x'$ 
 $| \text{tok}_{white} \Rightarrow y_2.\text{fail}; (x' \leftarrow \text{Bank}_A[\gamma] \leftarrow y_1, y_2, w_1, w_2); x \leftarrow x') @ \langle \text{guest}, \text{guest} \rangle$ 
 $\Psi; \cdot \vdash \text{Customer}_A :: y:\text{customer}(\psi, \text{guest})$ 
 $y \leftarrow \text{Customer}_A \leftarrow \cdot =$ 
 $y.\text{tok}_{white}; \text{case } y \text{ (succ} \Rightarrow (y' \leftarrow \text{Customer}_A[\gamma] \leftarrow \cdot); y \leftarrow y'$ 
 $| \text{fail} \Rightarrow (y' \leftarrow \text{Customer}_A[\gamma] \leftarrow \cdot); y \leftarrow y') @ \langle \psi, \text{guest} \rangle$ 
 $\Psi; \cdot \vdash \text{Statement}_A :: u:\text{transfer}(\psi, \psi)$ 
 $u \leftarrow \text{Statement}_A \leftarrow \cdot = u.\text{transaction}; \text{close } u @ \langle \psi, \psi \rangle$ 

```

■ **Figure 9** Security-polymorphic process definitions.

i.e., Customer_B , and Statement_B , would be similar. The example is typed using the security theory Ψ , consisting of the concrete security lattice Ψ_0 , the security variables ψ and ψ' , and the set of constraints $\{\text{guest} \sqsubseteq \psi \sqsubseteq \text{bank}, \text{guest} \sqsubseteq \psi' \sqsubseteq \text{bank}\}$. (See TR for the formal definition of a security theory.) To execute this program using the dynamics in Fig. 8, we provide the order-preserving substitution $\Psi_0 \Vdash \gamma' :: \Psi$, defined as $\gamma' := \{\psi \mapsto \text{alice}, \psi' \mapsto \text{bob}\}$.

Let us examine the pattern checks $\sim_{\langle \psi, \text{guest} \rangle}$ invoked by $\text{case } y_1(\dots)$ in Bank_A , relating the branches corresponding to black and white tokens. The sends along y_1 match in both branches, as demanded by SNDLAB (since $\Psi \not\Vdash \psi \sqsubseteq \text{guest}$), even though the sent labels are not the same. The unsynchronized spawn and send along w_1 is verified by UNSYNC-SPAWN₁ and UNSYNC₁, resp., since $\Psi \Vdash \psi \sqsubseteq \psi$. The matching tail calls are verified with SYNC-SPAWN.

6 Progress-sensitive noninterference

This section presents our main result, PSNI, which we prove using a logical relation.

6.1 Attacker model

The attacker model assumes a configuration \mathcal{D} with prior annotation of its free channels with security levels, the attacker's confidentiality level ξ , and a nondeterministic scheduler. The attacker knows the source code of \mathcal{D} , can only observe the messages sent along the free channels of \mathcal{D} with confidentiality level $c \sqsubseteq \xi$, and cannot measure the passing of time.

6.2 Noninterference via an integrity logical relation

Noninterference amounts to a process equivalence up to the confidentiality level ξ of an observer. In a message-passing system, it boils down to an equivalence of a configuration with interacting processes. This section focuses on noninterference for tree-shaped configurations. The definition can be extended to forests by enforcing pairwise relation between their trees.

An open configuration $\Psi_0; \Delta \Vdash \mathcal{D} :: x_\alpha:A(c, e)$ has the free channels Δ and x_α to communicate with its external environment; it sends outgoing messages to and receives incoming messages from the environment along these free channels. Two observationally equivalent

$$\begin{aligned}
 (\mathcal{B}_1, \mathcal{B}_2) \in \mathcal{E}_{\Psi_0}^\xi [\Delta \Vdash K]^{m+1} &\quad \text{iff} \quad (\mathcal{D}_1; \mathcal{D}_2) \in \text{Tree}_{\Psi_0}(\Delta \Vdash K) \text{ and } \forall \Upsilon_1, \Theta_1, \mathcal{D}'_1. \text{ if } \mathcal{D}_1 \xrightarrow{*_{\Upsilon_1, \Theta_1}} \mathcal{D}'_1 \text{ then} \\
 &\quad \exists \Upsilon_2 \mathcal{D}'_2. \text{ such that } \mathcal{D}_2 \xrightarrow{*_{\Upsilon_2}} \mathcal{D}'_2 \text{ and } \Upsilon_1 \subseteq \Upsilon_2 \text{ and} \\
 &\quad \forall y_\alpha \in \text{Out}(\Delta \Vdash K). \text{ if } y_\alpha \in \Upsilon_1. \text{ then } (\mathcal{D}'_1; \mathcal{D}'_2) \in \mathcal{V}_{\Psi_0}^\xi [\Delta \Vdash K]^{m+1}_{;y_\alpha} \text{ and} \\
 &\quad \forall y_\alpha \in \text{In}(\Delta \Vdash K). \text{ if } y_\alpha \in \Theta_1. \text{ then } (\mathcal{D}'_1; \mathcal{D}'_2) \in \mathcal{V}_{\Psi_0}^\xi [\Delta \Vdash K]^{m+1}_{y_\alpha;} \\
 (\mathcal{B}_1, \mathcal{B}_2) \in \mathcal{E}_{\Psi_0}^\xi [\Delta \Vdash K]^0 &\quad \text{iff} \quad (\mathcal{D}_1; \mathcal{D}_2) \in \text{Tree}_{\Psi_0}(\Delta \Vdash K)
 \end{aligned}$$

Figure 10 Term interpretation of logical relation.

- (l1) $(\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{V}_{\Psi_0}^\xi [\Delta, y_\alpha:1 \langle c, e \rangle \Vdash K]^{m+1}_{y_\alpha};$
 iff $(\mathcal{D}_1; \mathcal{D}_2) \in \text{Tree}_{\Psi_0}(\Delta, y_\alpha:1 \langle c, e \rangle \Vdash K) \text{ then}$
 $(\text{msg}(\text{close } y_\alpha^{\langle c, e \rangle}) \mathcal{D}_1; \text{msg}(\text{close } y_\alpha^{\langle c, e \rangle}) \mathcal{D}_2) \in \mathcal{E}_{\Psi_0}^\xi [\Delta \Vdash K]^m$
- (l2) $(\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{V}_{\Psi_0}^\xi [\Delta, y_\alpha : \oplus \{\ell: A_\ell\}_{\ell \in I} \langle c, e \rangle \Vdash K]^{m+1}_{y_\alpha};$
 iff $(\mathcal{D}_1; \mathcal{D}_2) \in \text{Tree}_{\Psi_0}(\Delta, y_\alpha : \oplus \{\ell: A_\ell\}_{\ell \in I} \langle c, e \rangle \Vdash K) \text{ and } \forall k_1, k_2 \in I. \text{ if } (c \sqsubseteq \xi \rightarrow k_1 = k_2) \text{ then}$
 $(\text{msg}(y_\alpha^{\langle c, e \rangle}.k_1) \mathcal{D}_1; \text{msg}(y_\alpha^{\langle c, e \rangle}.k_2) \mathcal{D}_2) \in \mathcal{E}_{\Psi_0}^\xi [\Delta, y_{\alpha+1}:A_{k_1} \langle c, e \rangle \Vdash K]^m$
- (l3) $(\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{V}_{\Psi_0}^\xi [\Delta, y_\alpha : \& \{\ell: A_\ell\}_{\ell \in I} \langle c, e \rangle \Vdash K]^{m+1}_{;y_\alpha}$
 iff $(\mathcal{D}_1; \mathcal{D}_2) \in \text{Tree}_{\Psi_0}(\Delta, y_\alpha : \& \{\ell: A_\ell\}_{\ell \in I} \langle c, e \rangle \Vdash K) \text{ and } \exists k_1, k_2 \in I. (c \sqsubseteq \xi \rightarrow k_1 = k_2) \text{ and}$
 $\mathcal{D}_1 = \text{msg}(y_\alpha^{\langle c, e \rangle}.k_1) \mathcal{D}'_1 \text{ and } \mathcal{D}_2 = \text{msg}(y_\alpha^{\langle c, e \rangle}.k_2) \mathcal{D}'_2 \text{ and}$
 $(\mathcal{D}'_1; \mathcal{D}'_2) \in \mathcal{E}_{\Psi_0}^\xi [\Delta, y_{\alpha+1}:A_{k_1} \langle c, e \rangle \Vdash K]^m$
- (l4) $(\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{V}_{\Psi_0}^\xi [\Delta, y_\alpha:A \otimes B \langle c, e \rangle \Vdash K]^{m+1}_{y_\alpha};$
 iff $(\mathcal{D}_1; \mathcal{D}_2) \in \text{Tree}_{\Psi_0}(\Delta, y_\alpha:A \otimes B \langle c, e \rangle \Vdash K) \text{ and } \forall x_\beta \notin \text{dom}(\Delta, y_\alpha:A \otimes B \langle c, e \rangle, K).$
 $(\text{msg}(\text{send } x_\beta^{\langle c, e \rangle}, y_\alpha^{\langle c, e \rangle}) \mathcal{D}_1; \text{msg}(\text{send } x_\beta^{\langle c, e \rangle}, y_\alpha^{\langle c, e \rangle}) \mathcal{D}_2) \in \mathcal{E}_{\Psi_0}^\xi [\Delta, x_\beta:A \langle c, e \rangle, y_{\alpha+1}:B \langle c, e \rangle \Vdash K]^m$
- (l5) $(\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{V}_{\Psi_0}^\xi [\Delta', \Delta'', y_\alpha:A \multimap B \langle c, e \rangle \Vdash K]^{m+1}_{;y_\alpha}$
 iff $(\mathcal{D}_1; \mathcal{D}_2) \in \text{Tree}_{\Psi_0}(\Delta', \Delta'', y_\alpha:A \multimap B \langle c, e \rangle \Vdash K) \text{ and}$
 $\mathcal{D}_1 = \mathcal{T}_1 \text{msg}(\text{send } x_\beta^{\langle c, e \rangle}, y_\alpha^{\langle c, e \rangle}) \mathcal{D}'_1 \text{ and for } \mathcal{T}_1 \in \text{Tree}_{\Psi_0}(\Delta' \Vdash x_\beta:A \langle c, e \rangle)$
 $\mathcal{D}_2 = \mathcal{T}_2 \text{msg}(\text{send } x_\beta^{\langle c, e \rangle}, y_\alpha^{\langle c, e \rangle}) \mathcal{D}'_2 \text{ and for } \mathcal{T}_2 \in \text{Tree}_{\Psi_0}(\Delta' \Vdash x_\beta:A \langle c, e \rangle) \text{ and}$
 $(\mathcal{T}_1; \mathcal{T}_2) \in \mathcal{E}_{\Psi_0}^\xi [\Delta' \Vdash x_\beta:A \langle c, e \rangle]^m \text{ and}$
 $(\mathcal{D}'_1; \mathcal{D}'_2) \in \mathcal{E}_{\Psi_0}^\xi [\Delta'', y_{\alpha+1}:B \langle c, e \rangle \Vdash K]^m$

Figure 11 Value interpretation of logical relation for left communications.

configurations may only differ in outgoing messages of confidentiality level $c_o \not\sqsupseteq \xi$, assuming that the incoming messages of confidentiality level $c_i \sqsubseteq \xi$ are the same. We introduce a *logical relation* that captures this idea and accounts for integrity and regrading policies.

The logical relation relates two open configurations \mathcal{D}_1 and \mathcal{D}_2 – the two runs of the program under consideration – and asserts that \mathcal{D}_1 and \mathcal{D}_2 send related messages to the environment, if they receive related messages from the environment. The term interpretation of the logical relation, defined in Fig. 10, allows the first configuration \mathcal{D}_1 to step internally until the configuration is ready to send or receive a message across at least one external channel. Then, it requires the second configuration \mathcal{D}_2 to step internally so that the resulting configurations are in the value interpretation of the logical relation, defined in Fig. 11 and Fig. 12. We call the external channels, e.g., $\Delta \Vdash K$ in Fig. 10, the interface of \mathcal{D}_1 and \mathcal{D}_2 . The metavariable K stands for either $x_\alpha:A \langle c, e \rangle$ or simply $_ : 1 \langle \top, \top \rangle$ which refers to an arbitrary unobservable channel.

The idea is to build an interface consisting of those external channels of the configurations that may impact the attacker’s observations. As such, not only do we need to include the observable channels, i.e., with confidentiality level $c \sqsubseteq \xi$, in the interface, but also those with higher integrity than the observer, i.e., with integrity level $e \sqsubseteq \xi$. After all, if a channel’s integrity is high enough (and thus its level is low), the messages along it may affect an observable outcome via synchronization patterns. We call such an interface *integrity interface* since low-confidentiality channels are all high-integrity by typing.

- (r₁) $(\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{V}_{\Psi_0}^{\xi} [\cdot \Vdash y_{\alpha}:1\langle c, e \rangle]_{y_{\alpha}}^{m+1}$
iff $(\mathcal{D}_1; \mathcal{D}_2) \in \text{Tree}_{\Psi_0}(\cdot \Vdash y_{\alpha})$ and $\mathcal{D}_1 = \text{msg}(\text{close } y_{\alpha}^{\langle c, e \rangle})$ and $\mathcal{D}_2 = \text{msg}(\text{close } y_{\alpha}^{\langle c, e \rangle})$
- (r₂) $(\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{V}_{\Psi_0}^{\xi} [(\Delta \Vdash y_{\alpha} : \oplus \{\ell:A_{\ell}\}_{\ell \in I}\langle c, e \rangle)]_{y_{\alpha}}^{m+1}$
iff $(\mathcal{D}_1; \mathcal{D}_2) \in \text{Tree}_{\Psi_0}(\Delta \Vdash y_{\alpha} : \oplus \{\ell:A_{\ell}\}_{\ell \in I}\langle c, e \rangle)$ and $\exists k_1, k_2 \in I. (c \sqsubseteq \xi \rightarrow k_1 = k_2)$
 $\mathcal{D}_1 = \mathcal{D}'_1 \text{msg}(y_{\alpha}^{\langle c, e \rangle}.k_1)$ and $\mathcal{D}_2 = \mathcal{D}'_2 \text{msg}(y_{\alpha}^{\langle c, e \rangle}.k_2)$
and $(\mathcal{D}'_1; \mathcal{D}'_2) \in \mathcal{E}_{\Psi_0}^{\xi} [\Delta \Vdash y_{\alpha+1}:A_{k_1}\langle c, e \rangle]^m$
- (r₃) $(\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{V}_{\Psi_0}^{\xi} [\Delta \Vdash y_{\alpha} : \& \{\ell:A_{\ell}\}_{\ell \in I}\langle c, e \rangle]_{y_{\alpha}}^{m+1}$
iff $(\mathcal{D}_1; \mathcal{D}_2) \in \text{Tree}_{\Psi_0}(\Delta \Vdash y_{\alpha} : \& \{\ell:A_{\ell}\}_{\ell \in I}\langle c, e \rangle)$ then $\forall k_1, k_2 \in I. \text{if } (c \sqsubseteq \xi \rightarrow k_1 = k_2) \text{ then}$
 $(\mathcal{D}_1 \text{msg}(y_{\alpha}^{\langle c, e \rangle}.k_1), \mathcal{D}_2 \text{msg}(y_{\alpha}^{\langle c, e \rangle}.k_2)) \in \mathcal{E}_{\Psi_0}^{\xi} [\Delta \Vdash y_{\alpha+1}:A_{k_1}\langle c, e \rangle]^m$
- (r₄) $(\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{V}_{\Psi_0}^{\xi} [\Delta', \Delta'' \Vdash y_{\alpha}:A \otimes B\langle c, e \rangle]_{y_{\alpha}}^{m+1}$
iff $(\mathcal{D}_1; \mathcal{D}_2) \in \text{Tree}_{\Psi_0}(\Delta', \Delta'' \Vdash y_{\alpha}:A \otimes B\langle c, e \rangle)$ and $\exists x_{\beta}.$
 $\mathcal{D}_1 = \mathcal{D}'_1 \mathcal{T}_1 \text{msg}(\text{send } x_{\beta}^{\langle c, e \rangle} y_{\alpha}^{\langle c, e \rangle})$ for $\mathcal{T}_1 \in \text{Tree}_{\Psi_0}(\Delta'' \Vdash x_{\beta}:A\langle c, e \rangle)$ and
 $\mathcal{D}_2 = \mathcal{D}'_2 \mathcal{T}_2 \text{msg}(\text{send } x_{\beta}^{\langle c, e \rangle} y_{\alpha}^{\langle c, e \rangle})$ for $\mathcal{T}_2 \in \text{Tree}_{\Psi_0}(\Delta'' \Vdash x_{\beta}:A\langle c, e \rangle)$ and
 $(\mathcal{T}_1; \mathcal{T}_2) \in \mathcal{E}_{\Psi_0}^{\xi} [\Delta'' \Vdash x_{\beta}:A\langle c, e \rangle]^m$ and
 $(\mathcal{D}'_1; \mathcal{D}'_2) \in \mathcal{E}_{\Psi_0}^{\xi} [\Delta' \Vdash y_{\alpha+1}:B\langle c, e \rangle]^m$
- (r₅) $(\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{V}_{\Psi_0}^{\xi} [\Delta \Vdash y_{\alpha}:A \multimap B\langle c, e \rangle]_{y_{\alpha}}^{m+1}$
iff $(\mathcal{D}_1; \mathcal{D}_2) \in \text{Tree}_{\Psi_0}(\Delta \Vdash y_{\alpha}:A \multimap B\langle c, e \rangle)$ and $\forall x_{\beta} \notin \text{dom}(\Delta, y_{\alpha}:A \multimap B\langle c, e \rangle).$
 $(\mathcal{D}_1 \text{msg}(\text{send } x_{\beta}^{\langle c, e \rangle} y_{\alpha}^{\langle c, e \rangle}); \mathcal{D}_2 \text{msg}(\text{send } x_{\beta}^{\langle c, e \rangle} y_{\alpha}^{\langle c, e \rangle})) \in \mathcal{E}_{\Psi_0}^{\xi} [\Delta, x_{\beta}:A\langle c, e \rangle \Vdash y_{\alpha+1}:B\langle c, e \rangle]^m$

Figure 12 Value interpretation of logical relation for *right* communications.

$(\Delta_1 \Vdash \mathcal{D}_1 :: x_{\alpha}:A_1\langle c_1, e_1 \rangle) \equiv_{\xi}^{\Psi_0} (\Delta_2 \Vdash \mathcal{D}_2 :: y_{\beta}:A_2\langle c_2, e_2 \rangle)$ iff
 $\mathcal{D}_1 \in \text{Tree}(\Delta_1 \Vdash x_{\alpha}:A_1\langle c_1, e_1 \rangle)$ and $\mathcal{D}_2 \in \text{Tree}(\Delta_2 \Vdash y_{\beta}:A_2\langle c_2, e_2 \rangle)$ and $\Delta_1 \Downarrow^{\text{ig}} \xi = \Delta_2 \Downarrow^{\text{ig}} \xi = \Delta$ and
 $x_{\alpha}:A_1\langle c_1, e_1 \rangle \Downarrow^{\text{ig}} \xi = y_{\beta}:A_2\langle c_2, e_2 \rangle \Downarrow^{\text{ig}} \xi = K$ and $\forall \mathcal{B}_1 \in \mathbf{L-IP}^{\xi}(\Delta_1). \forall \mathcal{B}_2 \in \mathbf{L-IP}^{\xi}(\Delta_2).$
 $\forall \mathcal{T}_1 \in \mathbf{L-IC}^{\xi}(x_{\alpha}:A_1\langle c_1, e_1 \rangle). \forall \mathcal{T}_2 \in \mathbf{L-IC}^{\xi}(y_{\beta}:A_2\langle c_2, e_2 \rangle).$
 $\forall m. (\mathcal{B}_1 \mathcal{D}_1 \mathcal{T}_1, \mathcal{B}_2 \mathcal{D}_2 \mathcal{T}_2) \in \mathcal{E}_{\Psi_0}^{\xi} [\Delta \Vdash K]^m$, and $\forall m. (\mathcal{B}_2 \mathcal{D}_2 \mathcal{T}_2, \mathcal{B}_1 \mathcal{D}_1 \mathcal{T}_1) \in \mathcal{E}_{\Psi_0}^{\xi} [\Delta \Vdash K]^m$.

$\cdot \in \mathbf{L-IP}^{\xi}(\cdot)$
 $\mathcal{B} \in \mathbf{L-IP}^{\xi}(\Delta, x_{\alpha}:A\langle c, e \rangle)$ iff
 $e \not\sqsubseteq \xi$ and $\mathcal{B} = \mathcal{B}' \mathcal{T}$ and $\mathcal{B}' \in \mathbf{L-IP}^{\xi}(\Delta)$ and $\mathcal{T} \in \text{Tree}(\cdot \Vdash x_{\alpha}:A\langle c, e \rangle)$, or
 $e \sqsubseteq \xi$ and $\mathcal{B} \in \mathbf{L-IP}^{\xi}(\Delta)$

$\mathcal{T} \in \mathbf{L-IC}^{\xi}(x_{\alpha}:A\langle c, e \rangle)$ iff
 $e \not\sqsubseteq \xi$ and $\mathcal{T} \in \text{Tree}(x_{\alpha}:A\langle c, e \rangle \Vdash _) : 1\langle \top, \top \rangle)$, or $e \sqsubseteq \xi$ and $\mathcal{T} = \cdot$

Figure 13 Logical equivalence.

To build an *integrity interface* for \mathcal{D}_1 and \mathcal{D}_2 , we close off their external low-integrity ($e \not\sqsubseteq \xi$) channels on the left by composing the channels with any well-typed provider and on the right with any well-typed client. We may use different low-integrity clients and providers to compose with each program run. These clients/providers can send different and unsynchronized messages along their high-confidentiality and low-integrity connections to \mathcal{D}_1 and \mathcal{D}_2 . The term interpretation is designed to ensure that well-typed configurations do not leak these different messages to the attacker. Fig. 13 defines an equivalence relation between two configurations based on this idea: it composes them with low-integrity providers/clients and calls the term interpretation symmetrically on the compositions. In the definition, we use the projection function to build the integrity interface, e.g., $\Delta \Downarrow^{\text{ig}} \xi$ projects out the channels $y_{\beta}:A\langle c, e \rangle \in \Delta$ with $\xi \not\sqsubseteq e$. The predicate $\mathcal{D}_1 \in \text{Tree}_{\Psi_0}(\Delta \Vdash K)$ indicates that the configuration \mathcal{D}_1 is well-typed. In the term and value interpretations, we generalize this predicate to the binary case, $(\mathcal{D}_1; \mathcal{D}_2) \in \text{Tree}_{\Psi_0}(\Delta \Vdash K)$ indicating that both \mathcal{D}_1 and \mathcal{D}_2 are of the same type.

The term interpretation allows stepping configuration $\mathcal{D}_1 \mapsto^{*\Upsilon_1:\Theta_1} \mathcal{D}'_1$ by iterated application of the rewriting rules defined in Fig. 8. The star expresses that zero to multiple internal steps can be taken. The superscripts $\Upsilon_1; \Theta_1$ denote two sets of channels occurring in the

interface $\Delta \Vdash K$. The set Θ_1 collects the *incoming* channels, i.e., channels that a process in \mathcal{D}_1 is ready to receive from, and the set Υ_1 collects the *outgoing* channels, i.e., channels with a message in \mathcal{D}_1 ready to be sent. Assuming that \mathcal{D}_1 steps to \mathcal{D}'_1 , generating the outgoing channels Υ_1 , \mathcal{D}_2 must be stepped $\mathcal{D}_2 \xrightarrow{*_{\Upsilon_2}} \mathcal{D}'_2$ to produce at least the same set of outgoing channels, i.e., the set Υ_2 such that $\Upsilon_1 \subseteq \Upsilon_2$. The term interpretation then calls the value interpretation on the resulting configurations $\mathcal{D}'_1, \mathcal{D}'_2$ for every channel that has a message ready for transmission in \mathcal{D}'_1 , and thus \mathcal{D}'_2 , and for every channel that has a process waiting for a message in \mathcal{D}'_1 . Insisting on Υ_2 being a superset of Υ_1 ensures progress-sensitive noninterference without timing attacks: if a configuration produces observable messages along a set of channels, the other configuration has to be able to produce the equivalent set of messages with zero or some internal steps. The term interpretation uses focus channels as a subscript to the value interpretation to support simultaneous communications – when there are multiple messages ready to be sent or received along channels in the interface. The subscript $\cdot; y_\alpha$ indicates that $y_\alpha \in \Upsilon_1$ and $y_\alpha; \cdot$ that $y_\alpha \in \Theta_1$.

The value interpretation accounts for every message sent from or received by \mathcal{D}_1 and \mathcal{D}_2 , amounting to two cases per connective: one for a message exchanged along a channel in K and one for a message exchanged along a channel in Δ . We refer to the former as communications to the *right* (Fig. 12) and the latter as communications to the *left* (Fig. 11). The value interpretation generally establishes the following pattern: it asserts relatedness of outgoing messages, but assumes relatedness of incoming messages. For example, $\&$ on the left (l_3 in Fig. 11) *asserts* the sending of related messages and pushes the messages into the environment, yielding $\mathcal{D}'_1, \mathcal{D}'_2$. Now, $\mathcal{D}'_1, \mathcal{D}'_2$, can each step internally, e.g., to consume the incoming messages, requiring them to be in the term interpretation. On the other hand, $\&$ on the right (r_3 in Fig. 12) *assumes* receipt of related messages and pushes the messages into the configurations \mathcal{D}_1 and \mathcal{D}_2 . *Relatedness* for messages is determined by how they can impact the attacker’s observations. If their carrier channel is observable to the attacker, i.e., has confidentiality level $c \sqsubseteq \xi$, then related messages must have the same labels. But if the channel only affects the attacker’s observations via synchronization patterns, related messages may have different labels. The clause $c \sqsubseteq \xi \rightarrow k_1 = k_2$ in the value interpretation conveys this, enforcing equality of the communicated labels only if the channel is observable.

Relatedness for higher-order types (\otimes and \multimap) is a bit more subtle. In particular, it requires future observations along the exchanged channels to be related. For example, l_5 in Fig. 11 for \multimap -left asserts existence of a message $\mathbf{msg}(\mathbf{send}x_\beta^{(c,e)} y_\alpha^{(c,e)})$ and of subtrees \mathcal{T}_1 and \mathcal{T}_2 in \mathcal{D}_1 and \mathcal{D}_2 . The clause comprises two invocations of the term relation, $(\mathcal{T}_1; \mathcal{T}_2) \in \mathcal{E}_{\Psi_0}^\xi [\Delta' \Vdash x_\beta : A(c, e)]^m$, asserting that future observations to be made along the sent channel x_β are related, and $(\mathcal{D}'_1; \mathcal{D}'_2) \in \mathcal{E}_{\Psi_0}^\xi [\Delta'', y_{\alpha+1} : B(c, e) \Vdash K]^m$, asserting that the continuations \mathcal{D}'_1 and \mathcal{D}'_2 are related. Conversely, r_5 in Fig. 12 for \multimap -right assumes receipt of a message $\mathbf{msg}(\mathbf{send}x_\beta^{(c,e)} y_\alpha^{(c,e)})$ and invokes the term relation $(\mathcal{D}_1 \mathbf{msg}(\mathbf{send}x_\beta^{(c,e)} y_\alpha^{(c,e)}); \mathcal{D}_2 \mathbf{msg}(\mathbf{send}x_\beta^{(c,e)} y_\alpha^{(c,e)})) \in \mathcal{E}_{\Psi_0}^\xi [\Delta, x_\beta : A(c, e) \Vdash y_{\alpha+1} : B(c, e)]^m$.

As we support general recursive types, we need an index to stratify our logical relation [3,5]. We tie our index to the number of *observations* that can be made along the interface $\Delta \Vdash K$, as suggested in [7]. We thus bound the value and term interpretation of our logical relation by the number of observations m , for $m \geq 0$, and attach them as superscripts to the relation’s interface $\Delta \Vdash K$. The base case of the term interpretation, i.e., $m = 0$, is a trivial relation.

The fundamental theorem states that any well-typed SINTEGRITY configuration is equivalent to itself up to the level of an arbitrary observer.

► **Theorem 1** (Fundamental theorem). *For all security levels ξ , and a well-typed configuration $\Psi_0; \Delta \Vdash \mathcal{D} :: u_\alpha : T(c, e)$ we have $(\Delta \Vdash \mathcal{D} :: u_\alpha : T(c, e)) \equiv_{\xi^0} (\Delta \Vdash \mathcal{D} :: u_\alpha : T(c, e))$.*

Proof. We present a proof sketch; see the TR for details. By the definition of logical equivalence (Fig. 13), we first need to close off the external low integrity ($e \not\sqsubseteq \xi$) channels in Δ and $u_\alpha:T\langle c, e \rangle$ by composing \mathcal{D} with arbitrary well-typed providers and clients, resp. We use low integrity clients, \mathcal{T}_1 and \mathcal{T}_2 , and low-integrity providers, \mathcal{B}_1 and \mathcal{B}_2 , to compose with each run, resulting in two configurations, $\mathcal{D}_1 = \mathcal{B}_1 \mathcal{D} \mathcal{T}_1$ and $\mathcal{D}_2 = \mathcal{B}_2 \mathcal{D} \mathcal{T}_2$. Configurations \mathcal{D}_1 and \mathcal{D}_2 are both well-typed for the integrity interface: $\Psi_0; \Delta' \Vdash \mathcal{D}_i :: K$ where $\Delta' = \Delta \Downarrow^{\text{ig}} \xi$ and $K = u_\alpha:T\langle c, e \rangle \Downarrow^{\text{ig}} \xi$. By the definition in Fig. 13, it is enough to show that \mathcal{D}_1 and \mathcal{D}_2 are in the term interpretation with the integrity interface, i.e., $\forall m. (\mathcal{D}_1, \mathcal{D}_2) \in \mathcal{E}_{\Psi_0}^\xi [\Delta' \Vdash K]^m$ and $\forall m. (\mathcal{D}_2, \mathcal{D}_1) \in \mathcal{E}_{\Psi_0}^\xi [\Delta' \Vdash K]^m$. We prove the former by induction on m ; the proof of the latter is symmetric. Specifically, we prove a more general theorem (Thm. 6.1 in TR) for any \mathcal{D}_1 and \mathcal{D}_2 with the same observable outcome, using the notion of relevant nodes (Def. 4.2 in TR). By the definition of the term interpretation in Fig. 10, the base case ($m = 0$) is straightforward. For the inductive case, following the first row of Fig. 10, we assume arbitrary Υ_1 , Θ_1 , and \mathcal{D}'_1 such that $\mathcal{D}_1 \mapsto^{*\Upsilon_1, \Theta_1} \mathcal{D}'_1$. We apply a lemma (Lem. 4.5 in TR) stating that \mathcal{D}_2 can simulate the internal steps taken by \mathcal{D}_1 , producing at least the same set of outgoing channels Υ_2 , i.e., $\mathcal{D}_2 \mapsto^{*\Upsilon_2} \mathcal{D}'_2$, such that \mathcal{D}'_1 and \mathcal{D}'_2 continue to have the same observable outcomes. Finally, for every channel x_α in Υ_1 and Θ_1 , we case analyze on the type of x_α , showing that \mathcal{D}'_1 and \mathcal{D}'_2 are in the value interpretation, with x_α being the focus channel. To do so, we use the induction hypothesis to establish that after the corresponding communication with the environment, the continuations of \mathcal{D}'_1 and \mathcal{D}'_2 are related by the term interpretation for a smaller index. \blacktriangleleft

6.3 Adequacy

Next, we prove an adequacy theorem showing that two logically equivalent configurations are *bisimilar* up to observations of confidentiality ξ .

For adequacy, we are interested in a *confidentiality interface*, i.e., channels with observable max confidentiality $c \sqsubseteq \xi$; after all, our goal is to prove that the configurations are equivalent up to the confidentiality of an observer. Because the integrity interface of our logical relation is a *superset* of the confidentiality interface, we need to close off those channels in the integrity interface that are of high-confidentiality ($c \not\sqsubseteq \xi$). Note that these high-confidentiality channels are of high-integrity ($e \sqsubseteq \xi$). To close off these channels, we compose the open configurations with high-confidentiality clients and providers, possibly different ones for each program run. These high-integrity clients and providers are connected to the open configurations via high-integrity channels and, as a result, may affect the observable outcome of the two runs via synchronization patterns. We therefore require them to be logically equivalent.

Based on this idea, Fig. 14 defines the *bisimulation up to* confidentiality ξ denoted as \approx_a^ξ , for two well-typed configurations: it first composes the two configurations with high-confidentiality providers (**Hrel-IPProvider**) and clients (**H-CClient**), while insisting that the high-integrity parts of the providers and clients are logically equivalent (using the relations **Hrel-IPProvider** and **Hrel-ICClient**). Then it invokes an asynchronous bisimulation \approx_a on the compositions. The definition uses a projection function $\Downarrow^{\text{cf}} \xi$ to build the confidentiality interface, e.g., $\Delta \Downarrow^{\text{cf}} \xi$ projects out the channels in Δ with confidentiality $c \not\sqsubseteq \xi$.

The *asynchronous bisimulation* \approx_a invoked by the definition in Fig. 14 uses a labeled transition system (LTS) following the standard definition of asynchronous bisimulation [38]. The relation $\mathcal{D}_1 \approx_a \mathcal{D}_2$ states that every internal step or external action of \mathcal{D}_1 can be (weakly) simulated by \mathcal{D}_2 and vice-versa. For example, when \mathcal{D}_1 takes an action by sending output q via an external channel x_α , i.e., $\mathcal{D}_1 \xrightarrow{x_\alpha q} \mathcal{D}'_1$, the bisimulation ensures that for some \mathcal{D}'_2 , we have $\mathcal{D}_2 \xrightarrow{x_\alpha q} \mathcal{D}'_2$ and $\mathcal{D}'_1 \approx_a \mathcal{D}'_2$. Here, $\xrightarrow{x_\alpha q}$ stands for taking zero or more internal steps before outputting q along x_α . The full definition of bisimulation is in the TR.

$$\begin{aligned}
 & \Delta_1 \Vdash \mathcal{D}_1 :: x_\alpha:A_1\langle c_1, e_1 \rangle \approx_a^\xi \Delta_2 \Vdash \mathcal{D}_2 :: y_\beta:A_2\langle c_2, e_2 \rangle \text{ iff} \\
 & \mathcal{D}_1 \in \text{Tree}(\Delta_1 \Vdash x_\alpha:A_1\langle c_1, e_1 \rangle) \text{ and } \mathcal{D}_2 \in \text{Tree}(\Delta_2 \Vdash y_\beta:A_2\langle c_2, e_2 \rangle) \text{ and} \\
 & \Delta = \Delta_1 \Downarrow^{\text{cf}} \xi = \Delta_2 \Downarrow^{\text{cf}} \xi \text{ and } K = y_\beta:A_2\langle c_2, e_2 \rangle \Downarrow^{\text{cf}} \xi = x_\alpha:A_1\langle c_1, e_1 \rangle \Downarrow^{\text{cf}} \xi \text{ and} \\
 & \Delta' = \Delta_1 \Downarrow^{\text{ig}} \xi = \Delta_2 \Downarrow^{\text{ig}} \xi \text{ and } K' = y_\beta:A_2\langle c_2, e_2 \rangle \Downarrow^{\text{ig}} \xi = x_\alpha:A_1\langle c_1, e_1 \rangle \Downarrow^{\text{ig}} \xi \text{ and} \\
 & \forall \mathcal{B}_1 \in \mathbf{H-CProvider}^\xi(\Delta_1), \mathcal{B}_2 \in \mathbf{H-CProvider}^\xi(\Delta_2). \\
 & \forall \mathcal{T}_1 \in \mathbf{H-CCClient}^\xi(x_\alpha:A_1\langle c_1, e_1 \rangle), \mathcal{T}_2 \in \mathbf{H-CCClient}^\xi(y_\beta:A_2\langle c_2, e_2 \rangle). \\
 & \text{if } (\mathcal{B}_1, \mathcal{B}_2) \in \mathbf{Hrel-IProvider}^\xi(\Delta' \setminus \Delta) \text{ and } (\mathcal{T}_1, \mathcal{T}_2) \in \mathbf{Hrel-IClient}^\xi(K' \setminus K) \text{ then } \mathcal{B}_1 \mathcal{D}_1 \mathcal{T}_1 \approx_a \mathcal{B}_2 \mathcal{D}_2 \mathcal{T}_2. \\
 & \cdot \in \mathbf{H-CProvider}^\xi(\cdot) \\
 & \mathcal{B} \in \mathbf{H-CProvider}^\xi(\Delta, x_\alpha:A\langle c, e \rangle) \text{ iff} \\
 & \quad c \not\sqsubseteq \xi \text{ and } \mathcal{B} = \mathcal{B}' \mathcal{T} \text{ and } \mathcal{B}' \in \mathbf{H-CProvider}^\xi(\Delta) \text{ and } \mathcal{T} \in \text{Tree}(\cdot \Vdash x_\alpha:A\langle c, e \rangle), \text{ or} \\
 & \quad c \sqsubseteq \xi \text{ and } \mathcal{B} \in \mathbf{H-CProvider}^\xi(\Delta) \\
 & \mathcal{T} \in \mathbf{H-CCClient}^\xi(x_\alpha:A\langle c, e \rangle) \text{ iff} \\
 & \quad c \not\sqsubseteq \xi \text{ and } \mathcal{T} \in \text{Tree}(x_\alpha:A\langle c, e \rangle \Vdash _) : 1(\top, \top), \text{ or} \\
 & \quad c \sqsubseteq \xi \text{ and } \mathcal{T} = \cdot \\
 & (\cdot, \cdot) \in \mathbf{Hrel-IProvider}^\xi(\cdot) \\
 & (\mathcal{B}_1, \mathcal{B}_2) \in \mathbf{Hrel-IProvider}^\xi(\Delta, x_\alpha:A\langle c, e \rangle) \text{ iff} \\
 & \quad e \not\sqsubseteq \xi \text{ and } \mathcal{B}_i = \mathcal{B}'_i \mathcal{T}_i \text{ and } (\mathcal{B}'_1, \mathcal{B}'_2) \in \mathbf{Hrel-IProvider}^\xi(\Delta), \text{ or} \\
 & \quad e \sqsubseteq \xi \text{ and } \mathcal{B}_i = \mathcal{B}'_i \mathcal{T}_i \text{ and } (\mathcal{B}'_1, \mathcal{B}'_2) \in \mathbf{Hrel-IProvider}^\xi(\Delta) \text{ and } \cdot \Vdash \mathcal{T}_1 \equiv_{\xi}^{\Psi_0} \mathcal{T}_2 :: x_\alpha:A\langle c, e \rangle. \\
 & (\cdot, \cdot) \in \mathbf{Hrel-IClient}^\xi(_, \langle \top, \top \rangle) \\
 & (\mathcal{T}_1, \mathcal{T}_2) \in \mathbf{Hrel-IClient}^\xi(x_\alpha:A\langle c, e \rangle) \text{ iff} \\
 & \quad e \not\sqsubseteq \xi \text{ or } e \sqsubseteq \xi \text{ and } x_\alpha:A\langle c, e \rangle \Vdash \mathcal{T}_1 \equiv_{\xi}^{\Psi_0} \mathcal{T}_2 :: _ : 1(\top, \top)
 \end{aligned}$$

Figure 14 Asynchronous bisimulation up to observations of confidentiality ξ .

Now we are ready to present our adequacy theorem stating that, given an observer level ξ , logically equivalent configurations are bisimilar up to observations of confidentiality ξ . The proof of the theorem relies on a compositionality lemma, which ensures a harmony between asserts and assumes in the value-interpretation of the logical relation.

Lemma 2 (Compositionality). $\forall m. (\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{E}_{\Psi_0}^\xi [\Delta, u_\alpha^{(c,e)} : T \Vdash K]^m$ and $\forall m. (\mathcal{T}_1; \mathcal{T}_2) \in \mathcal{E}_{\Psi_0}^\xi [\Delta' \Vdash u_\alpha^{(c,e)} : T]^m$ if and only if $\forall k. (\mathcal{T}_1 \mathcal{D}_1; \mathcal{T}_2 \mathcal{D}_2) \in \mathcal{E}_{\Psi_0}^\xi [\Delta', \Delta \Vdash K]^k$.

Theorem 3 (Adequacy). If $(\Delta_1 \Vdash \mathcal{D}_1 :: x_\alpha:A_1\langle c_1, e_1 \rangle) \equiv_{\xi}^{\Psi_0} (\Delta_2 \Vdash \mathcal{D}_2 :: y_\beta:A_2\langle c_2, e_2 \rangle)$ then $(\Delta_1 \Vdash \mathcal{D}_1 :: x_\alpha:A_1\langle c_1, e_1 \rangle) \approx_a^\xi (\Delta_2 \Vdash \mathcal{D}_2 :: y_\beta:A_2\langle c_2, e_2 \rangle)$.

Proof. Recall from Fig. 14 that \approx_a^ξ composes \mathcal{D}_1 and \mathcal{D}_2 with arbitrary high-confidentiality ($c \not\sqsubseteq \xi$) clients and providers, building a confidentiality interface $[\Delta^c \Vdash K^c]$. Let us call the high-confidentiality providers \mathcal{B}_1 and \mathcal{B}_2 and the high-confidentiality clients \mathcal{T}_1 and \mathcal{T}_2 . We can partition the providers into high-integrity and low-integrity parts to get $\mathcal{B}_1 = \mathcal{B}_1^{HI} \mathcal{B}_1^{LI}$ (similarly for the clients $\mathcal{T}_1 = \mathcal{T}_1^{HI} \mathcal{T}_1^{LI}$), where superscripts HI and LI correspond to high-integrity and low-integrtiy parts, resp. Our goal is to prove $\mathcal{B}_1^{HI} \mathcal{B}_1^{LI} \mathcal{D}_1 \mathcal{T}_1^{HI} \mathcal{T}_1^{LI} \approx_a \mathcal{B}_2^{HI} \mathcal{B}_2^{LI} \mathcal{D}_2 \mathcal{T}_2^{HI} \mathcal{T}_2^{LI}$.

Step 1. The first step is to show that the two compositions are related by the term interpretation as well, i.e., $\forall m. (\mathcal{B}_1^{HI} \mathcal{B}_1^{LI} \mathcal{D}_1 \mathcal{T}_1^{HI} \mathcal{T}_1^{LI}; \mathcal{B}_2^{HI} \mathcal{B}_2^{LI} \mathcal{D}_2 \mathcal{T}_2^{HI} \mathcal{T}_2^{LI}) \in \mathcal{E}[\Delta^c \Vdash K^c]$. To do so, we can use the definition from Fig. 13 for $\equiv_{\xi}^{\Psi_0}$ to compose \mathcal{D}_1 and \mathcal{D}_2 with given low integrity clients and providers $\mathcal{T}_1^{LI}, \mathcal{T}_2^{LI}, \mathcal{B}_1^{LI}$, and \mathcal{B}_2^{LI} to build the integrity interface $[\Delta^i \Vdash K^i]$ and get $\forall m. (\mathcal{B}_1^{LI} \mathcal{D}_1 \mathcal{T}_1^{LI}; \mathcal{B}_2^{LI} \mathcal{D}_2 \mathcal{T}_2^{LI}) \in \mathcal{E}[\Delta^i \Vdash K^i]$. However, this is not enough to achieve our goal as the relation pertains to the integrity interface, and thus, the composition only includes the low integrity providers/clients. To build the confidentiality interface and include the high integrity parts, we use the fact that the high integrity providers \mathcal{B}_1^{HI} and \mathcal{B}_2^{HI} (and clients \mathcal{T}_1^{HI} and \mathcal{T}_2^{HI}) are themselves logically equivalent. We use our compositionality lemma (Lem. 2) to compose the high-integrity channels in the integrity interface with these providers/clients and show that the composition results in two logically equivalent configurations, i.e., $\forall m. (\mathcal{B}_1^{HI} \mathcal{B}_1^{LI} \mathcal{D}_1 \mathcal{T}_1^{HI} \mathcal{T}_1^{LI}; \mathcal{B}_2^{HI} \mathcal{B}_2^{LI} \mathcal{D}_2 \mathcal{T}_2^{HI} \mathcal{T}_2^{LI}) \in \mathcal{E}[\Delta^c \Vdash K^c]$.

Step 2. We complete the proof by connecting our logically related configurations to an observational equivalence relation for session types [7], which is proved sound and complete for asynchronous bisimulation. We first show that our integrity term interpretation implies the observational equivalence relation in [7] when we consider a confidentiality interface, and then use their soundness result to show that the integrity term interpretation $\forall m. (\mathcal{B}_1^{HI} \mathcal{B}_1^{LI} \mathcal{D}_1 \mathcal{T}_1^{HI} \mathcal{T}_1^{LI}; \mathcal{B}_2^{HI} \mathcal{B}_2^{LI} \mathcal{D}_2 \mathcal{T}_2^{HI} \mathcal{T}_2^{LI}) \in \mathcal{E}[\Delta^c \Vdash K^c]$ implies bisimilarity $\mathcal{B}_1^{HI} \mathcal{B}_1^{LI} \mathcal{D}_1 \mathcal{T}_1^{HI} \mathcal{T}_1^{LI} \approx_a \mathcal{B}_2^{HI} \mathcal{B}_2^{LI} \mathcal{D}_2 \mathcal{T}_2^{HI} \mathcal{T}_2^{LI}$.

Next, we briefly explain how our integrity logical relation coincides with the observational equivalence relation (for well-typed configurations) in [7] when considering a confidentiality interface. The observational equivalence in [7] is defined via a logical relation similar to the one developed in this paper, but only considering the confidentiality interface. Let us call the term and value interpretations of our logical relation \mathcal{E}^i and \mathcal{V}^i (defined in Figs. 10–12) and the ones defined in [7] \mathcal{E}^c and \mathcal{V}^c , resp. The relation \mathcal{E}^i is invoked for the integrity interface $[\Delta^i \Vdash K^i]$ in Fig. 13, and similarly \mathcal{E}^c is invoked for the confidentiality interface $[\Delta^c \Vdash K^c]$, where, by definition, Δ^c is a subset of (or equal to) Δ^i and K^c is a subset of (or equal to) K^i . As the integrity logical relation may contain non-observable channels ($c \not\subseteq \xi$), it only insists that the same labels are sent when communication is along observable channels. Concretely, \mathcal{V}^i in lines (l_2) , (l_3) , (r_2) , and (r_3) only insists that the labels k_1 and k_2 sent/received along a channel are the same if $c \sqsubseteq \xi$. However, \mathcal{V}^c always enforces sending the same labels, since a priori the condition $c \sqsubseteq \xi$ holds for all the channels in its interface. In all other regards, \mathcal{E}^i and \mathcal{V}^i have the same definition as \mathcal{E}^c and \mathcal{V}^c . As a result, it is straightforward to observe that given an interface $\Delta^c \Vdash K^c$ with only observable channels (channels with $c \sqsubseteq \xi$), we have $\forall m. (\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{E}^i[\Delta^c \Vdash K^c]^m$ iff $\forall m. (\mathcal{D}_1; \mathcal{D}_2) \in \mathcal{E}^c[\Delta^c \Vdash K^c]^m$. ◀

7 Related work

IFC type systems using linearity. Conceptually most closely related to our work is the work by Zdancewic and Myers on *ordered linear continuations* [46, 47]. The authors consider continuation-passing style (CPS) security-typed languages to verify noninterference not only for source-level programs but also compiled code. The authors observe that the possibility to lower the `pc` label upon exiting control flow constructs, present in imperative source-level languages, is no longer available in a CPS language. To rectify the loss of flexibility they introduce ordered linear continuations. Similar to our pattern checks, ordered linear continuations allow downgrading of the `pc` label after branching on `high`, because linearity enforces the continuations to be used in every branch, in the order prescribed. In contrast to our work, the authors only consider a sequential language and only prove PINI. Our work moreover establishes the connection to integrity, facilitating regrading policies that are polymorphic in the security lattice for ultimate flexibility.

In another line of work Zdancewic and Myers again employ linearity for increased flexibility and a stronger noninterference statement [48]. The authors consider a concurrent language with a store and first-class channels. Their main focus is observational determinism, ensuring immunity to internal timing attacks and attacks that exploit information about thread scheduling. To this end the authors introduce linear channels and a race freedom analysis. Given that **SINTEGRITY** enjoys confluence, like other linear session type languages, it rules out timing attacks that exploit the relative order of messages, which seems to be a stronger property than immunity to internal timing attacks considered by the authors. Moreover, we establish PSNI for **SINTEGRITY**, whereas the authors only prove PINI.

IFC session type systems. In terms of underlying language, the work most closely related to ours is the one by Derakhshan et al. [7, 20]. The authors develop an IFC type system for the same family of linear session types but only consider confidentiality. Their system annotates the process term judgments with running and max confidentiality. Their typing rules only ensure that the running confidentiality (aka taint level) is updated correctly after each receive and that a tainted process does not leak information via send. In particular, the rules do not allow decreasing the taint level at any point. As a result the authors’ type system suffers from the same restrictiveness as other IFC type systems for concurrent languages, requiring each loop iteration to run at the maximal confidentiality reached throughout an arbitrary iteration. For example, the authors’ IFC type system rejects the banking example in § 5.4: as soon as the bank receives a message from one customer, say Alice, it will be tainted and cannot send a message to any other customer, say Bob. In fact, the authors’ IFC type system rejects all well-typed examples presented in this paper even though they enjoy PSNI. We make our IFC type system more flexible by designing synchronization policies to enable regrading of the taint level and using integrity labels to make the policies composable, both of which are novel to our system. Designing these composable policies was an intricate task, particularly due to dealing with both concurrency and general recursion.

Our logical relation for integrity is inspired by Balzer et al.’s [7] logical relation for equivalence. The logical relation for equivalence is defined based on the confidentiality interface. Our logical relation, however, is based on the larger integrity interface to enable the proof of the fundamental theorem. We prove our adequacy theorem by proving compositionality for our logical relation, which then allows us to recast our logical relation in terms of the logical relation for equivalence by the authors, delivering adequacy as a corollary.

IFC type systems for multiparty session types and process calculi. IFC type systems have also been explored for multiparty session types [10–13]. These works explore declassification [10, 12] and flexible runtime monitoring techniques [11, 13]. Our work not only differs in use of session type paradigm (i.e., binary vs. multiparty) but also in use of a logical relation for showing noninterference. Our work is more distantly related with IFC type systems for process calculi [16–18, 25, 25, 28, 29, 31, 34, 48]. These works prevent information leakage by associating a security label with channels/types/actions [29], read/write policies with channels [25, 25], or capabilities with expressions [16]. Honda et al. [29] also use a substructural type system and prove a sound embedding of Dependency Core Calculus (DCC) [2] into their calculus. Our work sets itself apart in its use of session types and meta theoretic developments based on logical relations. Moreover, our IFC type system is more permissive as it allows for regrading of the taint level, while preserving noninterference.

Declassification. Our notion of regrading may seem related to declassification, which has extensively been studied for IFC type systems for functional and imperative languages [1, 6, 15, 22, 32, 33, 44, 45, 49] and allows an entity to downgrade its level of confidentiality. However, our work significantly differs from declassification as it preserves PSNI, whereas declassification systems *deliberately* release information and thus intentionally *weaken* noninterference.

In particular, robust declassification [6, 15, 33, 44, 45, 49] prevents adversaries from exploiting downgrading of confidentiality, by complementing confidentiality with integrity. It uses integrity to ensure that downgrading decisions can be trusted, i.e., cannot be influenced by an attacker. As such, only high-integrity data can influence the taint level to be lowered. This is similar to our system, where the higher the integrity of a process, the lower level it can regrade the taint level. The difference, however, is that we enforce extra synchronization

policies on our high-integrity processes to ensure that they cannot induce information leaks by lowering the taint level. This contrasts with work on robust declassification, which introduces leakage intentionally and thus compromises noninterference.

References

- 1 Martín Abadi. Secrecy by typing in security protocols. In *TACS*, volume 1281 of *LNCS*, pages 611–638. Springer, 1997.
- 2 Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *POPL*, pages 147–160. ACM, 1999.
- 3 Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP*, volume 3924 of *LNCS*, pages 69–83. Springer, 2006.
- 4 Kaleb Alpernas, Cormac Flanagan, Sadjad Fouladi, Leonid Ryzhyk, Mooly Sagiv, Thomas Schmitz, and Keith Winstein. Secure serverless computing using dynamic information flow control. *Proc. ACM Program. Lang.*, 2(OOPSLA):118:1–118:26, 2018.
- 5 Andrew W. Appel and David A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS*, 23(5):657–683, 2001.
- 6 Aslan Askarov and Andrew C. Myers. Attacker control and impact for confidentiality and integrity. *Log. Methods Comput. Sci.*, 7(3), 2011.
- 7 Stephanie Balzer, Farzaneh Derakhshan, Robert Harper, and Yue Yao. Logical relations for session-typed concurrency. *CoRR*, abs/2309.00192, 2023. [arXiv:2309.00192](https://arxiv.org/abs/2309.00192).
- 8 Kenneth J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, Electronic Systems Division, Air Force Systems Command, United States Air Force, 1977.
- 9 Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, volume 6269 of *LNCS*, pages 222–236. Springer, 2010.
- 10 Sara Capecchi, Ilaria Castellani, and Mariangiola Dezani-Ciancaglini. Typing access control and secure information flow in sessions. *Inf. Comput.*, 238:68–105, 2014.
- 11 Sara Capecchi, Ilaria Castellani, and Mariangiola Dezani-Ciancaglini. Information flow safety in multiparty sessions. *Mathematical Structures in Computer Science*, 26(8):1352–1394, 2016. doi:10.1017/S0960129514000619.
- 12 Sara Capecchi, Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Tamara Rezk. Session types for access and information flow control. In *CONCUR*, pages 237–252, 2010.
- 13 Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Jorge A. Pérez. Self-adaptation and secure information flow in multiparty communications. *Formal Aspects Comput.*, 28(4):669–696, 2016.
- 14 Iliano Cervesato and Andre Scedrov. Relating state-based and process-based concurrency through linear logic. *Inf. Comput.*, 207(10):1044–1077, 2009.
- 15 Stephen Chong and Andrew C. Myers. Decentralized robustness. In *CSFW*, pages 242–256. IEEE, 2006.
- 16 Silvia Crafa, Michele Bugliesi, and Giuseppe Castagna. Information flow security for boxed ambients. *Electronic Notes in Theoretical Computer Science*, 66(3):76–97, 2002.
- 17 Silvia Crafa and Sabina Rossi. A theory of noninterference for the π -calculus. In *TGC*, volume 3705 of *LNCS*, pages 2–18. Springer, 2005.
- 18 Silvia Crafa and Sabina Rossi. Controlling information release in the pi-calculus. *Inf. Comput.*, 205(8):1235–1273, 2007.
- 19 Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *PLDI*, pages 50–63. ACM, 1999.
- 20 Farzaneh Derakhshan, Stephanie Balzer, and Limin Jia. Session logical relations for noninterference. In *LICS*, pages 1–14. IEEE, 2021.
- 21 Farzaneh Derakhshan, Stephanie Balzer, and Yue Yao. Regrading policies for flexible information flow control in session-typed concurrency. *CoRR*, 2024.

- 22 Elena Ferrari, Pierangela Samarati, Elisa Bertino, and Sushil Jajodia. Providing flexibility in information flow control for object-oriented systems. In *IEEE Symposium on Security and Privacy*, pages 130–140. IEEE, 1997.
- 23 Simon J. Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Informatica*, 42(2-3):191–225, 2005.
- 24 Daniel Heidin and Andrei Sabelfeld. A perspective on information flow control. Technical report, Marktoberdorf, 2011.
- 25 Matthew Hennessy. The security pi-calculus and non-interference. *J. Log. Algebraic Methods Program.*, 63(1):3–34, 2005.
- 26 Kohei Honda. Types for dyadic interaction. In *CONCUR*, volume 715 of *LNCS*, pages 509–523. Springer, 1993.
- 27 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.
- 28 Kohei Honda, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. Secure information flow as typed process behaviour. In *ESOP*, volume 1782 of *LNCS*, pages 180–199. Springer, 2000.
- 29 Kohei Honda and Nobuko Yoshida. A uniform type structure for secure information flow. In *POPL*, pages 81–92. ACM, 2002.
- 30 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *POPL*, pages 273–284. ACM, 2008.
- 31 Naoki Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Inf.*, 42(4):291–347, December 2005.
- 32 Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *TOSEM*, 9(4):410–442, 2000.
- 33 Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification and qualified robustness. *J. Comput. Secur.*, 14(2):157–196, 2006.
- 34 François Pottier. A simple view of type-secure information flow in the π -calculus. In *CSFW-15*, pages 320–330. IEEE, 2002.
- 35 Andrei Sabelfeld and Heiko Mantel. Static confidentiality enforcement for distributed programs. In *SAS*, volume 2477 of *LNCS*, pages 376–394. Springer, 2002.
- 36 Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE J. Sel. Areas Commun.*, 21(1):5–19, 2003.
- 37 Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *CSFW*, pages 200–214. IEEE, 2000.
- 38 Davide Sangiorgi and David Walker. *The π -calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.
- 39 Geoffrey Smith and Dennis M. Volpano. Secure information flow in a multi-threaded imperative language. In *POPL*, pages 355–364. ACM, 1998.
- 40 Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John C. Mitchell, and David Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *ICFP*, pages 201–214. ACM, 2012.
- 41 Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *ESOP*, volume 7792 of *LNCS*, pages 350–369. Springer, 2013.
- 42 Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2/3):167–188, 1996.
- 43 Philip Wadler. Propositions as sessions. In *ICFP*, pages 273–286. ACM, 2012.
- 44 Steve Zdancewic. A type system for robust declassification. In *MFPS*, volume 83 of *Electronic Notes in Theoretical Computer Science*, pages 263–277. Elsevier, 2003.
- 45 Steve Zdancewic and Andrew C. Myers. Robust declassification. In *CSFW*, pages 15–23. IEEE, 2001.

- 46 Steve Zdancewic and Andrew C. Myers. Secure information flow and CPS. In *ESOP*, volume 2028 of *LNCS*, pages 46–61. Springer, 2001.
- 47 Steve Zdancewic and Andrew C. Myers. Secure information flow via linear continuations. *High. Order Symb. Comput.*, 15(2-3):209–234, 2002.
- 48 Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *CSFW*, pages 1–15. IEEE, 2003.
- 49 Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *TOCS*, 20(3):283–328, 2002.

Mutation-Based Lifted Repair of Software Product Lines

Aleksandar S. Dimovski 

Mother Teresa University, Skopje, North Macedonia

Abstract

This paper presents a novel lifted repair algorithm for program families (Software Product Lines - SPLs) based on code mutations. The inputs of our algorithm are an erroneous SPL and a specification given in the form of assertions. We use variability encoding to transform the given SPL into a single program, called family simulator, which is translated into a set of SMT formulas whose conjunction is satisfiable iff the simulator (i.e., the input SPL) violates an assertion. We use a predefined set of mutations applied to feature and program expressions of the given SPL. The algorithm repeatedly mutates the erroneous family simulator and checks if it becomes (bounded) correct. Since mutating an expression corresponds to mutating a formula in the set of SMT formulas encoding the family simulator, the search for a correct mutant is reduced to searching an unsatisfiable set of SMT formulas. To efficiently explore the huge state space of mutants, we call SAT and SMT solvers in an incremental way. The outputs of our algorithm are all minimal repairs in the form of minimal number of (feature and program) expression replacements such that the repaired SPL is (bounded) correct with respect to a given set of assertions.

We have implemented our algorithm in a prototype tool and evaluated it on a set of `#ifdef`-based C programs (i.e., annotative SPLs). The experimental results show that our approach is able to successfully repair various interesting SPLs.

2012 ACM Subject Classification Software and its engineering → Software product lines; Theory of computation → Abstraction

Keywords and phrases Program repair, Software Product Lines, Code mutations, Variability encoding

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.12

Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact):*
<https://doi.org/10.4230/DARTS.10.2.5>

1 Introduction

A *program family* (Software Product Line - SPL) represents a set of similar programs, known as *variants*, generated from a common code base [2]. SPL engineering has been successfully applied in industry to meet the need for custom-tailored software. For instance, different variants from an SPL can target different platforms or may serve customization requirements for different customers. The variants are specified in terms of *features* selected for that particular variant. The popular `#ifdef` directives from the C preprocessor CPP [43] represent the most common way to implement such (annotative) program families. An `#ifdef` directive specifies under which presence conditions (i.e., feature selections or feature expressions), parts of code should be included or excluded from a variant at compile-time. SPLs are often used in the development of the embedded and safety-critical systems (e.g., mobile devices, cars, medicine, avionics), where their behavioral correctness is of primary interest. In particular, the focus is on applying various verification and analysis techniques from the field of formal methods, which can give stronger guarantees on the correctness of software systems. In the last decade, much effort has been invested in designing specialized so-called *lifted* (family-based) formal verification and analysis algorithms [4, 6, 9, 43, 30, 14, 23, 15, 20, 22, 25, 55], which allow simultaneous verification of all variants of an SPL in a single run by exploiting the commonalities between the variants. They usually return an error trace, which shows

 © Aleksandar S. Dimovski;
licensed under Creative Commons License CC-BY 4.0
38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 12; pp. 12:1–12:24



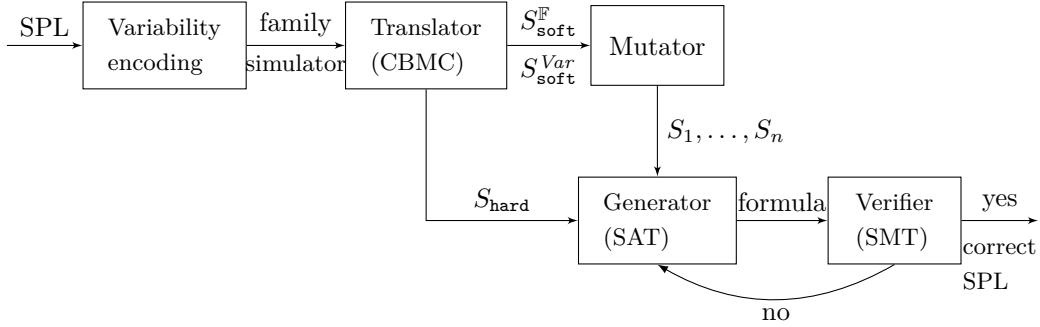


Figure 1 Diagram illustrating our lifted repair system.

how the given specification is violated. However, the users still need to process the obtained result, in order to isolate the cause of the error to a small part of the code and subsequently to repair the given SPL. Here, we consider the problem of *SPL repair*, which is defined to be a code transformation such that the repaired SPL satisfies a given specification (e.g. assertion). Automatic SPL repair is an important problem since even if an error is identified in the verification phase, the manual error-repair is a nontrivial time-consuming task that requires close knowledge of the SPL. For instance, the error-repair of one variant may cause new errors to appear in other variants due to the feature interaction in the given SPL [3]. Recently, researchers have developed several successful single-program repair tools [28, 37, 40, 42, 45, 46, 48, 50, 51]. However, these tools cannot be directly applied to SPLs as they are only able to handle pre-processed single programs.

In this paper, we lift the mutation-based approach ALLREPAIR [50, 51] for repairing single programs to program families (SPLs). Figure 1 illustrates our lifted repair system. More specifically, we use *variability encoding* [30, 56] to transform program families to single programs, called family simulators, by replacing compile-time variability with run-time variability (non-determinism). The (family) simulator, which contains the computations of all variants of a program family, is then translated into a set of SMT formulas using the CBMC bounded model checker [8]. The conjunction of the obtained SMT formulas is *satisfiable* iff there is an assertion violation in the given simulator iff there is an assertion violation in at least one variant of the original program family. On the other hand, the conjunction of the obtained SMT formulas is *unsatisfiable* iff all assertions are valid in the given simulator iff all assertions are valid in all variants of the original program family. We use a bounded notion of correctness, since we consider only bounded computations in which each loop and recursive call are inlined at most b times. Each statement in the simulator corresponds to a formula in the obtained set of SMT formulas, which can be partitioned into subset S_{hard} encoding parts of the program that cannot be changed and subset $S_{\text{soft}} = S_{\text{soft}}^{\text{F}} \cup S_{\text{soft}}^{\text{Var}}$ encoding parts of the program that can be changed. Therefore, mutating a feature or program expression found in a statement that can be changed corresponds to changing the respective SMT formula from $S_{\text{soft}}^{\text{F}}$ or $S_{\text{soft}}^{\text{Var}}$, respectively. The *mutator* unit generates mutated family simulators (mutants) by using a predefined set of syntactic mutations/edits applied to feature and program expressions. Hence, in our repair model, we permit feature and program expressions to be changed but not statements. For example, we allow replacement of `#ifdef` guards (e.g., by applying \neg to features, replacing \wedge/\vee with \vee/\wedge) and right-hand sides of assignments (e.g., by increasing or decreasing a constant, replacing $+/-$ with $-/+$). Thus, the size of the space of mutants depends on the choice of permissible mutations/edits used for repair. The mutants are explored in increasing number of mutations applied to the original family

simulator, so that only minimal sets of mutations are considered. Hence, the search in the space of mutants reduces to searching for an unsatisfiable set of SMT formulas. This search is performed using an iterative generate-and-verify process. The *generator* produces a minimally changed mutant using a SAT solver and the *verifier* checks the bounded-correctness of mutant using an SMT solver. This way, we find a solution with a minimal number of syntactic changes/edits to the original (incorrect) program family. Therefore, the type of errors that can be corrected is determined by the fixed set of syntactic mutations/edits, which can be applied to feature and program expressions. Hence, our approach can make repairs by replacing expressions in `#ifdef`-guards and right-hand sides of assignments with another expressions of the same form, but it cannot make repairs by replacing (adding/deleting) statements (e.g., replace assignment with if statement) or by replacing expressions with another expressions of different form (e.g., replace expression 5 with $x+y$). Both SAT and SMT solvers are used in an incremental way, which means that learned information is passed between successive calls. Since variants in a program family as well as mutated simulators are very similar, their encodings as sets of SMT formulas will have a lot in common. Hence, we can reuse the information that was gathered in checking previous mutated simulators to expedite the solution of the current one. The incremental solving was implemented via the mechanisms called assumptions and guard variables [26].

We have implemented our algorithm for repairing `#ifdef`-based C program families in a prototype tool, called SPLALLREPAIR, which is built on top of the ALLREPAIR tool [50, 51]. The tool uses the CBMC model checker [8] for translating single programs to SMT formulas, as well as the MINICARD [39] and Z3 [11] tools for SAT and SMT solving. We illustrate this approach for automatic repair on a number of C program families from the literature [10, 37, 46, 50, 51], and we report very encouraging results. We compare performances of two versions of our tool, with smaller and bigger sets of possible mutations, as well as with the **Brute-force** approach that repairs all variants from a program family one by one independently.

We summarize the contributions of this paper as follows:

Lifted Algorithm for SPL Repair: We propose a novel lifted algorithm based on variability encoding and syntactic code mutations for repairing program families;

Synthesizing Minimally Repaired SPLs: We automatically compute all minimal repaired program families (minimal in the number of code replacements) that are bounded correct by mutating feature and program expressions;

Implementation and Evaluation: We build a prototype tool for automatically repairing `#ifdef`-based C program families, and present experimental results by evaluating it on a dozen of C benchmarks.

2 Motivating Example

We now present an overview of our approach using a motivating example. Consider the `#ifdef`-based C program family `intro1`, shown in Fig. 2, which uses two Boolean features A and B. They induce a family of four variants defined by the set of configurations $\mathbb{K} = \{A \wedge B, \neg A \wedge B, A \wedge \neg B, \neg A \wedge \neg B\}$. For each configuration, a different variant (single program) can be generated by appropriately resolving `#if` directives. For example, the variant for configuration $(A \wedge B)$ will have both features A and B enabled (set to true or 1), thus yielding the body of `main()`: `int x=0; x=x+2; assert(x≥0); return x`. The variant for $(\neg A \wedge \neg B)$ will have both features A and B disabled (set to false or 0), so it has the following body of `main()`: `int x=0; x=x-2; assert(x≥0); return x`. In such program families, it may

```

int main(){
① int x := 0;
② #if (A) x := x+2; #endif
④ #if ( $\neg A \wedge \neg B$ ) x := x-2; #endif
⑥ assert (x  $\geq$  0);
⑦ return x;
}

```

Figure 2 intro1.

```

int A := [0, 1];
int B := [0, 1];
int main(){
    int x := 0;
    if (A) x := x+2;
    if ( $\neg A \wedge \neg B$ ) x := x-2;
    assert (x  $\geq$  0);
    return x;
}

```

Figure 3 intro2.

```

A0 := [0, 1];
B0 := [0, 1];
int main(){
    x0 := 0;
    g0 := A0;
    x1 := x0+2;
    x2 := g0?x1 : x0;
    g1 :=  $\neg A0 \wedge \neg B0$ ;
    x3 := x2-2;
    x4 := g1?x3 : x2;
    assert (x4  $\geq$  0);
    return x4;
}

```

Figure 4 intro3.

```

Sintro = {
    A0=[0, 1],
    B0=[0, 1],
    x0=0,
    g0=A0,
    x1=x0+2,
    x2=ite(g0,x1,x0),
    g1= $\neg A0 \wedge \neg B0$ ,
    x3=x2-2,
    x4=ite(g1,x3,x2),
     $\neg(x4 \geq 0)$ 
}

```

Figure 5 S_{intro} .

happen that errors (e.g., assertion violations) occur in some variants but not in others. In the `intro1` family, the assertion is valid for variants $(A \wedge B)$, $(A \wedge \neg B)$, $(\neg A \wedge B)$ since the returned value x will be 2, 2, 0, respectively. However, the assertion fails for variant $(\neg A \wedge \neg B)$ since the returned value x will be -2 in this case. The goal is to automatically repair this program family, so that the assertion is valid for all its variants.

If we make mutations only to feature expressions, there are two possible repairs of `intro1` that remedy the feature interaction $(\neg A \wedge \neg B)$ responsible for the fault. First, the feature expression (A) at loc. ② can be replaced with $(\neg A)$, thus making the assertion correct for all variants: the returned value x will be 0 for variants $(A \wedge B)$, $(A \wedge \neg B)$, $(\neg A \wedge B)$; and 2 for $(\neg A \wedge \neg B)$. Second, the feature expression $(\neg A \wedge \neg B)$ at loc. ④ can be replaced with $(A \wedge \neg B)$, thus making the assertion correct for all variants: the returned value x will be 0 for variants $(\neg A \wedge B)$, $(A \wedge \neg B)$, $(\neg A \wedge \neg B)$; and 2 for $(A \wedge B)$. If we make mutations only to program expressions, then one possible repair is the program expression $(x-2)$ at loc. ⑤ to be replaced with $(x+2)$. The above three repairs are all minimal patched mutations obtained by applying only one code mutation to the original program family. Note that the found repairs depend on the sets of mutations applied to feature and program expressions. For example, if we allow mutations of the arithmetic operator $-$ to $*$ and of the integer constant n to 0, we will also find additional minimal repairs that replace the expression $(x-2)$ at loc. ⑤ with $(x*2)$ or $(x-0)$.

Our algorithm for repairing program families goes through four steps. We refer to the running example `intro1` in Fig. 2 to demonstrate the steps.

- (1) We transform the program family to a single program, called family simulator, using variability encoding [30, 56], such that all features are first declared as global variables and non-deterministically initialized to 0 or 1, and then all `#if` directives are transformed into ordinary `if` statements with the same branch condition. For example, the single program `intro2` in Fig. 3 is a simulator for the program family `intro1` in Fig. 2. Features A and B are defined as non-deterministically initialized global variables and two `#if` directives are replaced with `if`-s.
- (2) The simulator is simplified (e.g., branch conditions are replaced with fresh Boolean variables), unwinded by unrolling loops and recursive functions b times, and converted to static single assignment (SSA) form. In the SSA form, time-stamped versions of program variables are created: every time a variable is assigned, the time-stamp is incremented by one and then the variable is renamed; every time a variable is read, it is renamed using the current time-stamp. Thus, the single program `intro3` in Fig. 4 is obtained by simplifying and converting to SSA form the simulator `intro2` in Fig. 3. For example, the `if` condition $(\neg A_0 \wedge \neg B_0)$ is assigned to a fresh Boolean variable g_1 , the first assignment to `x` is replaced by an assignment to x_0 , the second by an assignment to x_1 , etc. We use Φ -assignments to determine which copy of `x` will be used after `if`-s. For example, the Φ -assignment $x_2 := g_0?x_1 : x_0$ means that x_1 is used if g_0 is true, and x_0 is used if g_0 is false.
- (3) The simplified program in SSA form is converted to a program formula. Hence, the program `intro3` in Fig. 4 is converted to a set of SMT formulas S_{intro} shown in Fig. 5, such that the corresponding program formula φ_{intro} is a conjunction of all SMT formulas in S_{intro} . Note that the Φ -assignment $x_2 := g_0?x_1 : x_0$ is converted to the formula $x_2 = \text{ite}(g_0, x_1, x_0)$, which means $(g_0 \wedge x_2 = x_1) \vee (\neg g_0 \wedge x_2 = x_0)$, while `assert(be)` is converted to $(\neg \text{be})$. Therefore, a program is correct (i.e., all assertions in it are valid) iff the corresponding program formula is unsatisfiable.
- (4) By making mutations in the set of SMT formulas, we aim to construct an unsatisfiable program formula and report the corresponding program as repaired. In the running example, if one of the following mutations: $(g_0 = \neg A)$, $(g_1 = A \wedge \neg B)$, or $(x_3 = x_2 + 2)$, is applied to the set of SMT formulas S_{intro} in Fig. 5, we obtain an unsatisfiable program formula. This way, we generate a minimally mutated program family, which contains only one code mutation, that is correct.

3 Background

In this section, we introduce the background concepts used in later developments. We begin with the definition of syntax and semantics of program families. Then, we proceed to introducing the bounded program analysis for translating single programs to SMT formulas.

3.1 Program Families

Let $\mathbb{F} = \{A_1, \dots, A_n\}$ be a finite set of *Boolean features* available in a program family. A *configuration* $k : \mathbb{F} \rightarrow \{\text{true}, \text{false}\}$ is a truth assignment or a *valuation*, which gives a truth value to each feature. If $k(A) = \text{true}$, then feature A is enabled in configuration k , otherwise A is disabled. We assume that only a subset \mathbb{K} of all possible configurations are *valid*. Each configuration $k \in \mathbb{K}$ can also be represented by a formula: $(k(A_1) \cdot A_1 \wedge \dots \wedge k(A_n) \cdot A_n)$, where $\text{true} \cdot A = A$ and $\text{false} \cdot A = \neg A$. We write \mathbb{K} for the set of all valid configurations. We define *feature expressions*, denoted $\text{FeatExp}(\mathbb{F})$, as the set of propositional logic formulas over \mathbb{F} :

$$\theta (\theta \in \text{FeatExp}(\mathbb{F})) ::= \text{true} \mid A \in \mathbb{F} \mid \neg \theta \mid \theta \wedge \theta \mid \theta \vee \theta$$

We consider a simple sequential non-deterministic programming language, in which the program variables $Var = \{x_1, \dots, x_n\}$ are statically allocated and the only data type is the set \mathbb{Z} of mathematical integers. To define program families, a new compile-time conditional statement is introduced: “**#if** (θ) s **#endif**”, such that the statement s will be included in the variant corresponding to configuration $k \in \mathbb{K}$ only if θ is satisfied by k , i.e. $k \models \theta$. The syntax is:

$$\begin{aligned} s (s \in Stm) ::= & \text{skip} \mid x := ae \mid s; s \mid \text{if } (be) \text{ then } s \text{ else } s \mid \text{while } (be) \text{ do } s \mid \\ & \#if (\theta) s \#endif \mid \text{assert}(be) \mid \text{assume}(be) \\ ae (ae \in AExp) ::= & n \mid [n, n'] \mid x \mid ae \oplus ae, \\ be (be \in BExp) ::= & ae \bowtie ae \mid \neg be \mid be \wedge be \mid be \vee be \end{aligned}$$

where $n \in \mathbb{Z}$, $x \in Var$, $\oplus \in \{+, -, *, \%, /\}$, $\bowtie \in \{<, \leq, ==, !=\}$, and integer interval $[n, n']$ denotes a random integer in the interval. Without loss of generality, we assume that a program family P is a sequence of statements followed by a single assertion, whereas a single program p is a sequence of statements without **#if**-s followed by an assertion.

► **Remark 1.** The C preprocessor CPP [32] also uses other compile-time conditional statements that can be desugared and represented only by the **#if** construct we use in this work, e.g. **#if** (θ) s_0 **#else** s_1 **#endif** is translated into **#if** (θ) s_0 **#endif**; **#if** ($\neg\theta$) s_1 **#endif**. Compile-time conditional constructs can also be defined at the level of expressions, e.g. **#if** (θ) ae_0 **#else** ae_1 **#endif**, and they can be translated into compile-time conditional statements by code duplication [32]. We use variability at the level of statements for pedagogical reasons in order to keep the presentation focussed.

A program family is evaluated in two phases. First, the C preprocessor CPP [32] takes a program family s and a configuration $k \in \mathbb{K}$ as inputs, and produces a variant (single program without **#if**-s) corresponding to k as output. Second, the obtained variant is evaluated using the standard single-program semantics [20]. The first phase is specified by the projection function π_k , which is an identity for all basic statements and recursively pre-processes all sub-statements of compound statements. Hence, $\pi_k(\text{skip}) = \text{skip}$ and $\pi_k(s; s') = \pi_k(s); \pi_k(s')$. The most interesting case is “**#if** (θ) s **#endif**”, where the statement s is included in the variant k if $k \models \theta$;¹ otherwise s is excluded from the variant k . That is:

$$\pi_k(\#if (\theta) s \#endif) = \begin{cases} \pi_k(s) & \text{if } k \models \theta \\ \text{skip} & \text{if } k \not\models \theta \end{cases}$$

Given a program family P , the set of all variants derived from P is $\{\pi_k(P) \mid k \in \mathbb{K}\}$.

3.2 Bounded Program Analysis

Unbounded loops with memory allocation are the reason for the undecidability of the assertion verification problem [24]. To avoid undecidability, we impose a bound on the loops by discarding all executions paths in which a loop is iterated more than a pre-determined number of times. That is, we analyze a new bounded program that under-approximates the original program. Using such bounded program, we can build a SMT formula that represents its semantics. We now briefly explain how a pre-processed program without **#if**-s is translated into a set of SMT formulas using the CBMC bounded model checker [8]. We present only the details that are important to understand our algorithm.

¹ Since $k \in \mathbb{K}$ is a valuation function, either $k \models \theta$ holds or $k \not\models \theta$ holds for any θ .

The given pre-processed (single) program undergoes three transformations: simplification, unwinding, and conversion to SSA form. Recall from Section 2 that the simplification ensures that all branch conditions are replaced with fresh Boolean variables, whereas the SSA-form guarantees that each local variable has a single static point of definition. More specifically, in SSA-form each assignment to a variable x is changed into an unique assignment to a new variable x_i . Hence, if variable x has n assignments to it throughout the program, then n new variables x_0 to x_{n-1} are created to replace x . All uses of x are replaced by a use of some x_i . To decide which definition of a variable reaches a particular use after an `if`-statement with the guard g , we add the Φ -assignment $x_k := g?x_i : x_j$ after the `if`. This means that if control reaches the Φ -assignment via the path on which g is true, Φ selects x_i ; otherwise Φ selects x_j . This way, all uses of x after an Φ -assignment $x_k := g?x_i : x_j$ become uses of Φ -assignment x_k until the next assignment of x . The unwinding with bound b means that all `while` loops and recursive functions are unwound b times, so that we consider only so-called *b-bounded paths* that are going through them at most b times. For example, the statement “`while (be) do s`” after unwinding with $b = 2$ will be transformed to:

```
 $g:=be; \text{if } (g) \text{ then } \{s; g:=be; \text{if } (g) \text{ then } \{s; g:=be; \text{assume}(\neg g); \} \}$ 
```

where we use `assume($\neg g$)` to block all paths longer than the bound b . After the above three transformations, in the obtained simplified program all original expressions are right-hand sides (RHSs) of assignments, loops are replaced with `if`-s, and each variable is assigned once. For example, the simplified program `intro3` is obtained from `intro2` by the above three transformations.

The generated simplified program is converted to a set of SMT formulas S as follows. An assignment $x := ae$ is converted to equation formula $x = ae$; a Φ -assignment $x := be?x_1 : x_2$ is converted to formula $x = \text{ite}(be, x_1, x_2)$; an `assume(be)` is converted to formula be ; and an `assert(be)` is converted to formula $\neg be$. A statement that is part of a `while` body may be encoded by several formulas ϕ_1, \dots, ϕ_k in S due to the unwinding. In this case, we remove ϕ_1, \dots, ϕ_k from S , and add instead one conjunctive formula $(\phi_1 \wedge \dots \wedge \phi_k)$ in S . In effect, we obtain that one formula in S encodes a single statement in the original program. For example, the set S_{intro} is obtained from `intro3` by the above conversion.

The obtained set of formulas S is partitioned into three subsets: $S_{\text{soft}}^{\text{Var}}$ that contains all formulas corresponding to statements containing original program expressions, $S_{\text{soft}}^{\text{F}}$ that contains all formulas corresponding to statements containing original feature expressions, and S_{hard} that contains the other formulas corresponding to assertions, assumptions, Φ -assignments, and feature variable-assignments. Since all original program and feature expressions are RHSs of assignments after the simplification phase, all formulas in $S_{\text{soft}}^{\text{Var}}$ and $S_{\text{soft}}^{\text{F}}$ are either single assignment formulas ($x = ae$) or multiple assignment formulas $((x_1 = ae_1) \wedge \dots \wedge (x_k = ae_k))$. For example, the set S_{intro} in Fig. 5 is partitioned as follows:

$$\begin{aligned} S_{\text{soft}}^{\text{Var}} &= \{x_0=0, x_1=x_0+2, x_3=x_2-2\}, \\ S_{\text{soft}}^{\text{F}} &= \{g_0=A_0, g_1=\neg A_0 \wedge \neg B_0\}, \\ S_{\text{hard}} &= \{A_0=[0, 1], B_0=[0, 1], x_2=\text{ite}(g_0, x_1, x_0), x_4=\text{ite}(g_1, x_3, x_2), \neg(x_4 \geq 0)\} \end{aligned}$$

Given a pre-processed (single) program p , the program formula φ_p^b is the conjunction of all formulas in S , where b denotes the unwinding bound used in the transformation phase of p . The formula φ_p^b encodes all possible b -bounded paths in the program p that go through each loop at most b times. We say that a program p is *b-correct* if all assertions in it are valid in all b -bounded paths of p .

► **Proposition 2** ([8]). *A pre-processed (single) program p is b-correct iff φ_p^b is unsatisfiable.*

A satisfying assignment (model) of φ_p^b represents a b -bounded path of p that satisfies all assumptions but violates at least one assertion. In the following, we omit to write p and b in the program formula φ_p^b when they are clear from the context.

Our approach reasons about loops by unrolling them, so it is sensitive to the unrolling bound. We now present an example, where the unrolling bound has impact on the assertion validity.

► **Example 3.** Consider the program:

```
int i:=0, x:=0; while (i<3) do {i:=i+1; x:=x+1; }
```

Suppose that the assertion to be checked is `assert(x≥3)` at the final location. If we use the unrolling bound $b = 2$, we will find that the program is incorrect due to the spurious execution path that runs the `while`-body 2 times. Hence, we will needlessly try to repair this correct program. However, if we use the bound $b \geq 3$, then we will establish that the program is correct and so no repair is needed.

Suppose that the assertion to be checked is `assert(x<3)` at the final location. If we use the unrolling bound $b = 2$, we will find that the program is correct since the assertion is valid for all 2-bounded paths, so no repair will be performed. However, if we use the bound $b \geq 3$, then we will truly establish that the program is incorrect and so a repair is needed.

To enable incremental SMT solving, the program formula φ is instrumented with Boolean variables called *guard variables*. More specifically, a formula $\varphi = \phi_1 \wedge \dots \wedge \phi_n$ is replaced with $\varphi' = (x_1 \implies \phi_1) \wedge \dots \wedge (x_n \implies \phi_n)$, where x_1, \dots, x_n are fresh guard variables. In effect, the formula $(x_i \implies \phi_i)$ can be satisfied by setting x_i to false. Some guard variables called *assumptions* are conjuncted with φ' and passed to an incremental SMT solver. For example, $\varphi' \wedge x_1 \wedge x_2$ is satisfiable iff ϕ_1 and ϕ_2 are satisfiable, since the satisfying assignment will set x_3, \dots, x_n to false thus making $(x_3 \implies \phi_3), \dots, (x_n \implies \phi_n)$ true. Thus, an incremental SMT-solver checking the satisfiability of $\varphi' \wedge x_1 \wedge x_2$ will only check satisfiability of ϕ_1 and ϕ_2 , thus essentially disabling formulas ϕ_3, \dots, ϕ_n .

We will use formulas of the form `AtMost({l1, ..., ln}, k)` (resp., `AtLeast({l1, ..., ln}, k)`) to require that at most (resp., at least) k of the literals l_1, \dots, l_n are true. They are called *Boolean cardinality formulas* encoding that $\sum_{i=1}^n l_i \leq k$ (resp., $\sum_{i=1}^n l_i \geq k$), where l_i is a literal assigned the value 1 if true and the value 0 if false, and $k \in \mathbb{N}$. We will use the MINICARD SAT-solver [39] to check their satisfiability.

4 Lifted Repair Algorithm

In this section, we present our lifted repair algorithm, called SPLALLREPAIR, for repairing program families. We first give a high-level overview of the algorithm, and then describe its components more formally.

High-level Description

The SPLALLREPAIR is given in Algorithm 1. It takes as input a program family P , an unwinding bound b , and a repair size r that limits the search space to only mutated programs with at most r mutations (changes to the original code) applied at once. The algorithm goes through an iterative generate-and-verify procedure, implemented using an interplay between an SAT solver and an SMT solver. In particular, we use an SAT solver in the generate phase to find a mutated program from the search space, whereas we use an SMT solver in the verify phase to check if the mutated program is correct.

Algorithm 1 SPLAllRepair(P, b, r).

Input: Program family P , unwinding bound b , repair size r

Output: Set of solutions Sol

```

1  $p_{sim} := \text{VarEncode}(P)$  ;
2  $(S_{\text{hard}}, S_{\text{soft}}^{\text{Var}}, S_{\text{soft}}^{\mathbb{F}}) := \text{CBMC}(p_{sim}, b)$  ;
3  $(S_1, \dots, S_n) := \text{Mutate}(S_{\text{soft}}^{\text{Var}}, S_{\text{soft}}^{\mathbb{F}})$  ;
4  $(S'_1, \dots, S'_n, V_1, \dots, V_n, V_{\text{orig}}) := \text{InstGuardVars}(S_1, \dots, S_n)$  ;
5  $\varphi_{sim}^b := (\wedge_{s \in S_{\text{hard}}} s) \wedge (\wedge_{s \in S'_1 \cup \dots \cup S'_n} s)$  ;
6  $\varphi := (\wedge_{i=1}^n \text{AtMost}(V_i, 1)) \wedge (\wedge_{i=1}^n \text{AtLeast}(V_i, 1))$  ;
7  $k := 1$ ;  $Sol := \emptyset$  ;
8 while ( $k \leq n$ )  $\wedge$  ( $k \leq r$ ) do
9    $\varphi_k := \varphi \wedge \text{AtLeast}(V_{\text{orig}}, n - k)$  ;
10   $satres, V := \text{SAT}(\varphi_k)$  ;
11  if ( $satres$ ) then
12     $smtres := \text{IncrementalSMT}(\varphi_{sim}^b \wedge \wedge_{v \in V} v)$  ;
13    if ( $\neg smtres$ ) then
14       $Sol := Sol \cup V$  ;
15       $\varphi_k := \varphi_k \wedge (\vee_{v \in V \setminus (V_{\text{orig}})} \neg v)$  ;
16    else
17       $\varphi_k := \varphi_k \wedge (\vee_{v \in V} \neg v)$  ;
18  else
19     $k := k + 1$  ;
20 if ( $\text{Timeout}$ ) then return  $Sol$  ;
21 return  $Sol$ ;
```

The SPLALLREPAIR starts by generating the family simulator p_{sim} using the pre-processor **VarEncode** procedure (line 1). Then, the **CBMC** translation procedure calls the **CBMC** model checker to generate the triple $(S_{\text{hard}}, S_{\text{soft}}^{\text{Var}}, S_{\text{soft}}^{\mathbb{F}})$ of sets of formulas corresponding to p_{sim} as explained in Section 3.2 (line 2). By calling the **Mutate** procedure, we generate all possible mutations S_1, \dots, S_n of formulas in $S_{\text{soft}}^{\text{Var}}$ and $S_{\text{soft}}^{\mathbb{F}}$ (line 3). Here S_i is a set of formulas obtained by mutating some $\phi_i \in S_{\text{soft}}^{\text{Var}} \cup S_{\text{soft}}^{\mathbb{F}}$. Thus, S_1, \dots, S_n correspond to n program locations where an error may occur. Next, we use the **InstGuardVars** procedure to instrument all formulas in S_1, \dots, S_n by fresh guard variables, so that the results are sets of instrumented formulas S'_1, \dots, S'_n and sets of fresh guard variables V_1, \dots, V_n used to guard formulas in S'_1, \dots, S'_n (line 4). Here $S'_i = \{(x \implies \phi) \mid \phi \in S_i, x \text{ is a fresh guard variable}\}$. The set V_{orig} contains guard variables corresponding to original formulas in $S_{\text{soft}}^{\text{Var}}$ and $S_{\text{soft}}^{\mathbb{F}}$. The program formula φ_{sim}^b is then initialized to be the conjunction of all formulas from S_{hard} and all instrumented formulas from $S'_1 \cup \dots \cup S'_n$ (line 5). Subsequently, we search the space of all mutated formulas in increasing size order using the variable k , which is initialized to 1 and increased after each iteration (lines 8–20). In particular, we generate the boolean formula φ_k [13] (line 9) expressing that k guard variables are not original, that is $n - k$ are original (by using $\text{AtLeast}(V_{\text{orig}}, n - k)$), and there is exactly one guard variable selected for each statement in the program (by using $\varphi \equiv \wedge_{i=1}^n \text{AtMost}(V_i, 1) \wedge \wedge_{i=1}^n \text{AtLeast}(V_i, 1)$, line 6). This means that every satisfying assignment of φ_k represents one mutated program formula of size at most k (i.e. with k changes to the original code). The boolean formula φ_k is fed to an SAT solver, which can handle Boolean cardinality formulas, to check its satisfiability. If

φ_k is unsatisfiable, this means that there are no unexplored mutated program formulas of size k so we increase k by one (line 19) and generate a new formula φ_k . Otherwise, if φ_k is satisfiable, we store in a set V all guard variables assigned true in the given satisfying assignment of φ_k (line 10). To check the correctness of the mutated program corresponding to the satisfying assignment V of φ_k , we call an incremental SMT solver to check φ_{sim}^b with all guards in V passed as assumptions (i.e., $\varphi_{sim}^b \wedge \wedge_{v \in V} v$) (line 12). This is the same to checking the conjunction of all formulas in S_{hard} and all soft formulas guarded by variables in V , since all other soft formulas will get satisfied by setting their guard variables to false. Notice that SMT formulas solved consecutively in the iteration are very similar, thus sharing majority of their assumptions and all hard formulas. This means that most of what was learnt in solving the previous formula can be reused to solve the current one. If the result of incremental SMT solving is true, the mutated program is not correct so we block V from further exploration (line 17). Otherwise, we report V as a possible solution (i.e., a repaired program family) and block all supersets of V for further exploration (lines 14,15). The algorithm terminates when either the whole search space of mutated programs is inspected, i.e. all possible combinations of guard variables in n locations are explored as assumptions ($k > n$, line 8), or the subspace of mutated programs with at most r mutations is inspected ($k > r$, line 8), or a time limit is reached (line 20).

► **Example 4.** Let p be a simulator with 4 statements that can be mutated. Let p_1 be a repaired mutant of p consisting of mutating statement 1 with mutation M_1^1 (guard variable v_1^1) and statement 3 with mutation M_3^2 (guard variable v_3^2). Then blocking any superset of this mutation is done by adding the blocking clause $(\neg v_1^1 \vee \neg v_3^2)$ to the Boolean formula φ_k representing the search space of all mutants. This means do not apply either M_1^1 to statement 1 or do not apply M_3^2 to statement 3.

On the other hand, let p_2 be a buggy mutant of p consisting of mutating statement 1 with mutation M_1^2 (guard variable v_1^2) and statement 4 with mutation M_4^2 (guard variable v_4^2). The guards for original statements 2 and 3 are v_2^{orig} and v_3^{orig} . Then the blocking clause $(\neg v_1^2 \vee \neg v_2^{orig} \vee \neg v_3^{orig} \vee \neg v_4^2)$ will be added to prune from the search space exactly the mutant p_2 . Note that smaller blocking clause (with smaller number of literals) will result in a larger set of pruned mutants.

Pre-Processor: VarEncode

The aim of the pre-processor **VarEncode** procedure is to transform an input program family P with sets of features \mathbb{F} and configurations \mathbb{K} into an output pre-processed (single) program without **#if**-s, called family simulator. The set of configurations \mathbb{K} includes all possible combinations of feature values. In the pre-transformation phase, we convert each feature $A \in \mathbb{F}$ into the global variable A non-deterministically initialized to 0 or 1. Let $\mathbb{F} = \{A_1, \dots, A_n\}$ be the set of available *features* in the program family P . We generate the following pre-transformed program:

$$\text{pre-t}(P) \equiv \text{int } A_1 := [0, 1], \dots, A_n := [0, 1]; P$$

We now define a rewrite rule for eliminating **#if**-s from $\text{pre-t}(P)$. Let \mathbb{K} be the set of configurations in the family P that can be equated to a propositional formula $\kappa = \vee_{k \in \mathbb{K}} k$. Note that if \mathbb{K} contains all possible combinations of feature values, then $\kappa \equiv \text{true}$. The rewrite rule replaces **#if**-s with ordinary **if**-s whose guards are strengthened with the feature model κ .

$$\text{#if } (\theta) s \text{ #endif } \rightsquigarrow \text{if } (\theta \wedge \kappa) \text{ then } s \text{ else skip} \quad (\text{R-1})$$

If the current program family being transformed matches the abstract syntax tree node of the shape `#if (θ) s #endif`, then replace it with the RHS of rule (R-1). We write $\text{VarEncode}(P)$ to be the final transformed single program obtained by repeatedly applying rule (R-1) on $\text{pre-t}(P)$ and on its transformed versions until we reach a point at which this rule can no longer be applied.

A memory state $\sigma : \Sigma = \text{Var} \rightarrow \mathbb{Z}$ is a function mapping each program variable to a value. Given a single program p and a memory state σ , we write $\llbracket p \rrbracket \sigma$ for the set of final states that can be derived by executing all terminating paths (computations) of p starting in the input state σ . Note that the result is a set of states since our language is non-deterministic. We define $\llbracket p \rrbracket = \cup_{\sigma \in \mathcal{P}(\Sigma)} \llbracket p \rrbracket \sigma$ to be the set of final states that can be reached by p from any possible input state $\sigma \in \mathcal{P}(\Sigma)$ (where $\mathcal{P}(\Sigma)$ is the powerset of Σ). The following result shows that the set of final states from terminating computations of $\text{VarEncode}(P)$ coincides with the union of final states from terminating computations of all variants derived from the program family P .

► **Proposition 5** ([30]). *For a program family P , $\llbracket \text{VarEncode}(P) \rrbracket = \cup_{k \in \mathbb{K}} \llbracket \pi_k(P) \rrbracket$.*

► **Example 6.** Consider the program family `intro1` in Fig. 2 and its family simulator `intro2≡VarEncode(intro1)` in Fig. 3. The states σ contain only one program variable `x`. Hence, the semantics of all variants of `intro1` is:

$$\begin{aligned} \llbracket \pi_{A \wedge B}(\text{intro1}) \rrbracket &= [x \mapsto 2], \quad \llbracket \pi_{A \wedge \neg B}(\text{intro1}) \rrbracket = [x \mapsto 2] \\ \llbracket \pi_{\neg A \wedge B}(\text{intro1}) \rrbracket &= [x \mapsto 0], \quad \llbracket \pi_{\neg A \wedge \neg B}(\text{intro1}) \rrbracket = [x \mapsto -2] \end{aligned}$$

On the other hand, the semantics of `intro2≡VarEncode(intro1)` is:

$$\llbracket \text{VarEncode}(\text{intro1}) \rrbracket = \{[x \mapsto -2], [x \mapsto 0], [x \mapsto 2]\}$$

Mutate

As explained in Section 3.2, the SMT formulas in $S_{\text{soft}}^{\text{Var}}$ and $S_{\text{soft}}^{\mathbb{F}}$ correspond to statements containing program and feature expressions, so our goal is to repair the given erroneous program family by applying mutations to those formulas. A *mutation* is a replacement of a program/feature expression with another expression of the same type. For example, feature expressions A and $A \wedge B$ can be replaced with $\neg A$ and $(A \vee \neg B)$, while program expressions x and $x + 2$ can be replaced with 0 and $x - 2$. We maintain a fixed list of syntactic mutations for each type of program and feature expressions. Let us assume that mutations M_1, \dots, M_j can be applied to a formula $\phi \in S_{\text{soft}}^{\text{Var}} \cup S_{\text{soft}}^{\mathbb{F}}$. Then, $\text{Mutate}(\phi) = \{\phi, M_1(\phi), \dots, M_j(\phi)\}$. Finally, we have $\text{Mutate}(S_{\text{soft}}^{\text{Var}}, S_{\text{soft}}^{\mathbb{F}}) = \prod_{\phi \in S_{\text{soft}}^{\text{Var}} \cup S_{\text{soft}}^{\mathbb{F}}} \text{Mutate}(\phi)$.

We now present the variability-specific mutations applied to feature expressions: $A \rightarrow \neg A$ (read: feature A is replaced by $\neg A$) and $\neg A \rightarrow A$ for features $A \in \mathbb{F}$, as well as $\{\wedge, \vee\}$ (read: logical operator \wedge can be replaced with \vee , and vice versa).

► **Example 7.** Recall that $S_{\text{soft}}^{\mathbb{F}} = \{g0=A0, g1=\neg A0 \wedge \neg B0\}$ for our running example `intro1`. If we use the variability-specific mutations $A \rightarrow \neg A$, $\neg A \rightarrow A$ for $A \in \mathbb{F}$ and $\{\wedge, \vee\}$, we obtain:

$$\begin{aligned} \text{Mutate}(S_{\text{soft}}^{\mathbb{F}}) = \{g0=A0, g0=\neg A0, g1=\neg A0 \wedge \neg B0, g1=A0 \wedge \neg B0, g1=\neg A0 \wedge B0, \\ g1=A0 \wedge B0, g1=\neg A0 \vee \neg B0, g1=A0 \vee \neg B0, g1=\neg A0 \vee B0, g1=A0 \vee B0\} \end{aligned}$$

Post-Processor: Interpreting results

The solutions obtained by calling the ALLREPAIR tool to repair $\text{VarEncode}(P)$ are interpreted back on the original program family P . Any possible repair for $\text{VarEncode}(P)$, which consists of replacing some feature and program expressions, represents a valid repair for P as well.

This is due to the fact that our transformed program $\text{VarEncode}(P)$ contains all possible paths that may occur in any variant $\pi_k(P)$ for $k \in \mathbb{K}$. A single program (variant) is b -correct if it has no b -bounded path that leads to an assertion failure, while a program family is b -correct if all its variants are b -correct. Therefore, the b -correctness and possible repair of $\text{VarEncode}(P)$ and P are isomorphic (identical).

More formally, by using Propositions 2 and 5, we can prove the following result.

► **Corollary 8.** *Let P and b be a program family and an unwinding bound.*

- (i) $\varphi_{\text{VarEncode}(P)}^b$ is unsatisfiable iff $\forall k \in \mathbb{K}. \pi_k(P)$ is b -correct iff P is b -correct.
- (ii) $\varphi_{\text{VarEncode}(P)}^b$ is satisfiable iff $\exists k \in \mathbb{K}. \pi_k(P)$ is not b -correct iff P is not b -correct.

Correctness

We first use Corollary 8 to show the b -correctness of the SPLALLREPAIR algorithm (where b is the unwinding bound). That is, every solution returned by SPLALLREPAIR is minimal repaired program family (b -soundness), and every minimal repaired program family with respect to mutations we apply is eventually returned by SPLALLREPAIR (b -relative completeness). Our algorithm explores all mutated programs in increasing size order starting with size 1. Every returned solution is minimally repaired due to the fact that it would have been blocked by another smaller solution in a previous iteration. Therefore, the b -correctness (b -soundness and b -relative completeness) of SPLALLREPAIR follows from the b -correctness of ALLREPAIR shown in [50] and Corollary 8 (i.e., the fact that $\text{VarEncode}(P)$ and P are isomorphic with respect to b -correctness).

The SPLALLREPAIR always terminates, as there are only finitely many mutations that can be applied to any type of (feature and program) expressions so the algorithm enumerates all possible mutated programs (simulators) until it finds the minimal repaired ones if any. This way, we have proved the following result.

► **Theorem 9.** *The algorithm $\text{SPLAllRepair}(P, b, r)$ is b -bounded correct and terminates.*

5 Evaluation

We now evaluate our approach for mutation-based lifted repair of SPLs. We show that our approach can efficiently repair various interesting `#ifdef`-based C program families, and we compare the runtime performances and precision of two versions of our algorithm, with smaller and bigger sets of mutations, as well as with the `Brute-force` approach that repairs all variants of a program family one by one independently.

Implementation

We have implemented our lifted repair algorithm SPLALLREPAIR in a prototype tool, which is built on top of the ALLREPAIR tool [50, 51] for repairing single programs. The pre-processor `VarEncode` procedure is implemented in Java, while the translation and mutation procedures (`CBMC` and `Mutate` in Algorihtm 1) are implemented by modifying the CBMC model checker [8] written in C++, where variability-specific mutations are defined. Moreover, we have experimented by defining various mutations to other types of program expressions (see below). The repair phase is implemented by calling the ALLREPAIR tool [50] written in Python. We also call the MINICARD SAT solver [39] and the Z3 SMT solver [11]. The altered CBMC (plus $\sim 1K$ LOC) takes as input a family simulator, and generates a `gsmt2` file containing SMT formulas for all possible mutations of the corresponding statements in the input program. The ALLREPAIR ($\sim 2K$ LOC) takes as input a `gsmt2` file, generates formulas for SAT and SMT solving, and handles all calls to them.

The tool accepts programs written in C with `#ifdef/#if` directives. It uses three main parameters: *mutation level* that defines the kind of mutations that will be applied to feature and program expressions; *unwinding bound b* that shows how many times loops and recursive functions will be inlined; and *repair size r* that specifies how many mutations will be applied at most to buggy programs. We use two mutation levels: *level 1* contains simpler mutations that are often sufficient for repairment, while *level 2* contains all possible mutations we apply. For each type of feature and program expression, the list of syntactic mutations/edits in level 1 and level 2 is given below:

type of exp.	level 1	level 2
arithmetic op.	{+, -}, {*%, ÷}	{+, -, *, %, ÷}
relational op.	{<, ≤}, {>, ≥}, {==, !=}	{<, ≤, >, ≥, ==, !=}
logical op.	{&&, }	{&&, }
bit-wise op.	{>>, <<}, {&, , ^}	{>>, <<, &, , ^}
program vars		$x \rightarrow 0, x \rightarrow -x$
integer constants		$n \rightarrow n+1, n \rightarrow n-1, n \rightarrow -n, n \rightarrow 0$
feature vars	$A \rightarrow \neg A, \neg A \rightarrow A$	$A \rightarrow \neg A, \neg A \rightarrow A$

For example, for arithmetic operators in mutation level 1 we have two sets $\{+, -\}$ and $\{*%, ÷\}$, which means that $+$ is replaced with $-$ and vice versa, and $*, %, ÷$ can be replaced with each other. On the other hand, in mutation level 2 we have one set $\{+, -, *, %, ÷\}$, which means that any arithmetic operator from the set can be replaced with any other. Mutations on feature variables $A \in \mathbb{F}$ in both levels include negations of feature variables ($A \rightarrow \neg A, \neg A \rightarrow A$), whereas for program variables $x \in Var$ in level 2 we have mutations for replacing them with 0 ($x \rightarrow 0$) and changing the sign ($x \rightarrow -x$). Integer constants $n \in \mathbb{Z}$ in mutation level 2 can be increased by one, decreased by one, minused, or replaced with 0.

Experimental setup and Benchmarks

Experiments are run on 64-bit Intel®CoreTM i7-1165G7 CPU@2.80GHz, VM Ubuntu 22.04.3 LTS, with 8 GB memory. We use a timeout value of 400 sec. The implementation, benchmarks, and all obtained results are available from: <https://zenodo.org/records/11179373>. For the aim of evaluation, we ran: (1) our tool with mutation level 1, denoted SPLALLREPAIR₁; (2) our tool with mutation level 2, denoted SPLALLREPAIR₂; and (3) the **Brute-force** approach that uses a preprocessor to generate all variants of a program family and then applies the single-program repair tool ALLREPAIR to each individual variant independently.

The evaluation is performed on a dozen of C programs: two warming-up examples (`intro1` in Fig. 2 and `feat-inter` in Fig. 6); four commonly known algorithms (`feat_power` in Fig. 7, `factorial` in Fig. 8, `sum` in Fig. 9 and `sum_mton` in Fig. 10); `Codeflaws` [53], `TCAS` [29], and `Qclose` [10] benchmarks that are widely used for evaluating program repair tools [10, 37, 46, 50, 51]; as well as `MinePump` system [38] from the `product-lines` category of SV-COMP 2024 (<https://sv-comp.sosy-lab.org/2024>) that is often used to assess product-line verification in the SPL community [4, 9, 56, 55]. `Codeflaws` consists of programs taken from buggy user submissions to the programming contest site Codeforces (<http://codeforces.com>). For each program, there is a correct reference version and several buggy versions. Traffic Alert and Collision Avoidance System (`TCAS`) represents an aircraft collision detection system used by all US commercial aircrafts. The `TCAS` benchmark suite consists of a reference (correct) implementation and 41 faulty versions. In our experiments, we use 10 faulty versions that can be repaired using the mutations we apply in our approach. The

```

void main(){
    int x := 0;
    #if (A) x := x+2; #endif
    #if (B ∧ C) x := x-2; #endif
    assert(x ≥ 0 && x < 4);
}

```

Figure 6 feat-inter.

```

int feat_power(int n){
    assume(n ≥ 1);
    int res := 0;
    #if (¬A) int i := 1;
    #else int i := 0; #endif
    while(i < 3){
        res = res * n;
        i++;
    }
    #if (A) assert(sum == n * n * n);
    #else assert(sum == n * n * n * n); #endif
    return res;
}

```

Figure 7 feat_power.

```

void main(int n){
    assume(n ≥ 0);
    int res1 := fact(n);
    int res2 := fact_correct(n);
    assert(res1 == res2);
}
int fact_correct(int x){
    int res = 1;
    for (int i=2; i ≤ x; i++)
        res *= i;
    return res;
}

```

Figure 8 factorial.

```

int fact(int x){
    int res = 1, i = 2;
    while (#if (A) (i < x) #else (i ≤ x) #endif){
        res = mult(res, i);
        i++;
    }
    return res;
}
int mult(int x, int y){
    int res = 0;
    for (int i = 1; i ≤ y; i++)
        #if (B) res -= x; #else res += x; #endif
    return res;
}

```

Qclose benchmarks are used for evaluating the **Qclose** program repair tool [10], which consist of a reference (correct) implementation and several faulty versions for each programming task. In the case of **Codeflaws**, **TCAS**, and **Qclose**, we have selected several faulty versions of each benchmark and we have created a buggy program family out of them. For example, we use **tcas_v3** and **tcas_v12** (resp., **tcas_v16** and **tcas_v17**) to create the **tcas_spl1** (resp., **tcas_spl2**) program family. Then, we use assertions to check the equivalence of the results returned by the program family and the reference (correct) version (for example, see **main()** of **factorial** in Fig. 8). Note that the correct version is marked so that it will not be mutated. The **MinePump** SPL system contains 730 LOC and six independent optional features: **start**, **stop**, **methaneAlarm**, **methaneQuery**, **lowWaterSensor**, **highWaterSensor**. When activated, the controller should switch on the pump when the water level is high, but only if there is no methane in the mine. We consider two specifications of the **MinePump** system encoded as assertions in SV-COMP 2024: **MinePump_spec1** checks whether the pump is not running if the level of methane is critical; and **MinePump_spec3** checks whether the pump is running if the level of water is high. Table 1 presents characteristics of the benchmarks, such as: the file name (Benchmark), the number of features $|F|$ (note that $|K| = 2^{|F|}$), and the lines of code (LOC).

```
int sum(int n){
    assume(n ≥ 1);
    int sum := 0, i := 0;
    #if (A) i := 1; #endif
    while(i < n) {
        #if (B) sum+=i;
        #else sum-=i; #endif
        i++;
    }
    assert(sum==n*(n+1)/2);
    return sum;
}
```

Figure 9 sum.

```
int sum_mton(int n, int m){
    assume(n ≥ 1&&m ≥ 1);
    #if (A) assume(n ≥ m);
    #else assume(m ≥ n); #endif
    int sum := 0;
    #if (A) int i := n;
    #else int i := m; #endif
    while(#if (A) (i ≤ n)#else (i ≤ m) #endif)
    {
        sum:=sum-i;
        i++; }
    #if (A) assert(sum==(n*(n+1)-m*(m-1))/2);
    #else assert(sum==(m*(m+1)-n*(n-1))/2);
    #endif
    return sum; }
```

Figure 10 sum_mton.

Examples

We now present several of our examples in detail. Consider the program family `feat-inter` in Fig. 6. The error occurs due to the feature interaction ($\neg A \wedge B \wedge C$). In particular, the variant ($\neg A \wedge B \wedge C$) is: `int x=0; x=x-2; assert(x≥0 && x<4)`. So the assertion fails since `x` has value -2 at the assertion location. The simplest fix from mutation level 1, which replaces `x:=x-2` with `x:=x+2`, does not work as it introduces a new error in other variants. In this case, the feature interaction ($A \wedge B \wedge C$) causes the assertion failure since the value of `x` will be 4 at the assertion location for variant ($A \wedge B \wedge C$). Therefore, SPLALLREPAIR₁ reports that no repair is found by searching the space of 7 mutants in 0.254 sec. However, if we consider mutations of level 2 then SPLALLREPAIR₂ successfully finds a repair, which replaces `x:=x-2` with `x:=x-0`, by searching the space of 25 mutants in 0.315 sec. On the other hand, the `Brute-force` approach applies mutations to all faulty variants independently. As the only faulty variant is ($\neg A \wedge B \wedge C$), it will report the repair that replaces `x:=x-2` with `x:=x+2`. This is a correct repair for the variant ($\neg A \wedge B \wedge C$), but not for the entire family. This example shows that sometimes the `Brute-force` approach may not report correct results due to the feature interaction.

The program family `feat_power` in Fig. 7 should find the third power of `n` when feature `A` is enabled and the fourth power of `n` when `A` is disabled. SPLALLREPAIR₁ suggests fixes in 0.722 sec that replace the feature expression ($\neg A$) with (`A`) when initializing variable `i` and replace `while`-guard (`i < 3`) with (`i ≤ 3`). The `Brute-force` finds that variant (`A`) is correct, but variant ($\neg A$) is not correct and no fix is suggested as integer constants cannot be mutated in level 1. Some possible repairs of variant ($\neg A$) in level 2 will make variant (`A`) incorrect. For example, changing the `while`-guard to (`i ≤ 3`) will make variant (`A`) incorrect since it is initialized to 0 so it will return the fourth power of `n` instead of the third.

The program `factorial` in Fig. 8 contains two implementations of the factorial function: a correct one, called `fact_correct`, and a buggy one, called `fact`, that represents a program family with four variants. The assertion requires that the results returned from each variant of `fact` are equivalent with the result returned from `fact_correct`. We do not apply mutations to `fact_correct`, but only to the program family `fact`. All three approaches suggest fixes that replace the `while`-guard (`i < x`) with (`i ≤ x`) and the assignment `res-=x` with `res+=x`.

Table 1 Performance results of SPLALLREPAIR₁ vs. SPLALLREPAIR₂ vs. Brute-force. All times in sec.

Benchmarks	F	LOC	SPLALLREPAIR ₁			SPLALLREPAIR ₂			Brute-force		
			Fix	Space	Time	Fix	Space	Time	Fix	Space	Time
intro1	2	20	✓	7	0.252	✓	25	0.304	✓	5	0.981
feat-inter	3	20	✗	7	0.254	✓	25	0.315	✗	9	2.110
feat_power	1	20	✓	16	0.722	✓	403	7.79	✗	8	0.882
factorial	2	50	✓	86	2.540	✓	1603	107.3	✓	81	4.196
sum	2	30	✓	17	0.376	✓	266	2.656	✓	18	1.147
sum_mton	1	20	✓	32	0.770	✓	681	15.22	✗	10	0.556
4-A-Codeflaws	2	95	✗	52	0.426	✓	1390	2.578	✗	36	1.180
651-A-Codeflaws	2	85	✓	180	3.394	✓	2829	38.53	✓	237	5.78
tcas_spl1	1	305	✗	37	0.99	✓	158	6.10	✗	37	1.41
tcas_spl2	1	305	✗	38	1.19	✓	164	8.94	✗	38	1.47
Qlose_multiA	3	32	✗	122	0.711	✓	5415	69.21	✗	65	5.781
Qlose_iterPower	2	30	✗	9	0.973	✓	38	2.921	✗	16	1.391
MinePump_spec1	6	730	✓	38	300.0	✗	-	timeout	✓	-	timeout
MinePump_spec3	6	730	✓	39	291.0	✗	-	timeout	✓	-	timeout

Consider the program family `sum` in Fig. 9, which computes the sum of all integers from 0 to a given input integer `n`. The specification indicates that given a positive input `n` ($n \geq 1$), the output represented by the variable `sum` is $n*(n+1)/2$. The body of `sum` is implemented in an iterative fashion. There are two features `A` and `B` that enable different initializations of `i` and different updates of `sum`. Let us consider mutations of level 1. If the repair size is 1 (i.e., only one original expression can be mutated), our tool cannot find a repair by searching the space of 7 mutants in 0.321 sec. However, if the repair size is 2, then SPLALLREPAIR₁ suggests a fix that replaces the `while`-guard (`i < n`) with (`i ≤ n`) and the assignment `sum-=i` with `sum+=i`. The search space contains 17 mutants and the tool explores it in 0.376 sec. The Brute-force approach reports a correct repair in 1.147 sec.

The program family `sum_mton` in Fig. 10 computes the sum $m + (m + 1) + \dots + n$ when feature `A` is enabled and ($n \geq m$), and the sum $n + (n + 1) + \dots + m$ when feature `A` is disabled and ($m \geq n$). The corresponding specifications assert that the returned value `sum` is equal to $(n * (n + 1) - m * (m - 1))/2$ when `A` is on and $(m * (m + 1) - n * (n - 1))/2$ when `A` is off. The programmer has made two mistakes: when initializing variable `i` and when updating variable `sum` in the `while`-body. SPLALLREPAIR₁ suggests fixes in 0.770 sec that replace the feature expression (`A`) with ($\neg A$) when initializing variable `i` and replace `sum:=sum-i` with `sum:=sum+i` when updating `sum`. However, the Brute-force cannot fix any of the two variants since mutating variable `n` (resp., `m`) to other variable `m` (resp., `n`) is not allowed.

Performance

Table 1 shows performance results of running SPLALLREPAIR₁, SPLALLREPAIR₂, and the Brute-force approach on the given benchmarks. We use mutation level 1 for Brute-force. Note that the Brute-force approach calls translation, mutation, and repair procedures for each variant separately, whereas SPLALLREPAIR₁ and SPLALLREPAIR₂ call these procedures only once per program family. Moreover, the Brute-force approach can only

Table 2 Performance results of SPLALLREPAIR₁ for different values of the unwinding bound $b = 2, 5, 8$. All times in sec.

Benchmarks	$b = 2$		$b = 5$		$b = 8$	
	Fix	Time	Fix	Time	Fix	Time
feat_power	✗	0.254	✓	0.722	✓	0.978
factorial	✗	1.231	✓	3.540	✓	6.524
sum	✗	0.304	✓	0.376	✓	0.456
sum_mton	✗	0.589	✓	0.770	✓	0.922
651-A-Codelflaws	✓	1.814	✓	3.394	✓	6.828

find repairs by mutating program expressions. The default values for unwinding bound is $b = 5$ and for repair size is $r = 1$. However, for some benchmarks whose repaired versions contain more than one code mutation, we use the minimal value of repair size r that allows one approach to find a correct solution. For example, we use repair size $r = 2$ for `sum`. For each approach, there are three columns: “Fix” that specifies with ✓ (resp., ✗) whether the given approach finds (resp., does not find) a correct repair for a given benchmark; “Space” that specifies how many mutants have been inspected; and “Time” that specifies the total time (in seconds) needed for the given tasks to be performed.

From Table 1, we can see that SPLALLREPAIR₁ and SPLALLREPAIR₂ combined outperform the **Brute-force** approach with respect to both repairability and runtime. In particular, SPLALLREPAIR₂ fully repairs 12 benchmarks, which is better than 8 full correct repairs reported by SPLALLREPAIR₁ and 4 full correct repairs reported by the **Brute-force** approach that use the same mutations of level 1 (see also Discussion below). Note that SPLALLREPAIR₂ and the **Brute-force** timeout after 400 sec for the MinePump system. Hence, they report only a partial list of possible repairs, denoted by ✓. On the other hand, SPLALLREPAIR₁ achieves time speed-ups compared to **Brute-force** when report the same results, that range from 1.2 to 4 times. If we compare SPLALLREPAIR₁ and SPLALLREPAIR₂, we can see that there is a trade-off between repairability and runtime. That is, SPLALLREPAIR₂ is more precise (12 vs. 8 fixes) but slower (from 1.2 to 42 times slow-down when report the same results) compared to SPLALLREPAIR₁.

Table 2 shows performance results of running SPLALLREPAIR₁ on a selected set of benchmarks for different unwinding bounds b . Recall that our approach reasons about loops by unrolling (unwinding) them, so it is sensitive to the chosen unwinding bound. By choosing larger bounds b , we will obtain more precise results (more genuine repairs), but we will also obtain longer SMT formulas and slower speeds of the repairing tasks. We can see that the running times of all repairing tasks grow with the number of bound b . This is due to the fact that longer SMT formulas are generated, which need more time to be verified. Of course, we will also obtain more precise results for bigger values of b , and less precise results (i.e., some genuine repairs will not be reported) for smaller values of b . Hence, there is a precision/speed tradeoff when choosing the unwinding bound b . We obtain similar results for SPLALLREPAIR₂ and the **Brute-force**.

Discussion

In summary, our experiments demonstrate that our tool outperforms the **Brute-force** approach, and moreover it can be used for repairing various SPLs with different sizes of LOC, configuration space, and mutation space. Although SPLALLREPAIR₁ and **Brute-force**

have similar precision (8 vs. 4 fixes) due to the use of same sets of mutations, there is still a difference in the quality of the reported results. As we argued before, SPLALLREPAIR₁ and SPLALLREPAIR₂ report repaired program families obtained by fixing both feature and program expressions, whereas **Brute-force** only reports the repaired variants obtained by fixing program expressions. Hence, the results from **Brute-force** have to be analyzed by the user to produce information comparable to that returned by SPLALLREPAIR₁ and SPLALLREPAIR₂ in the form of repaired program families. Moreover, the fixes of individual variants may cause errors in other variants as evidenced by **feat-inter** and **feat_power**.

The main bottleneck for real-world SPLs, such as **MinePump** with 730 LOC and 6 features, is the huge space of mutants. The problem is that the search space of mutants grows very rapidly as the number of changeable expressions (statements) included in S_{soft} grows. For example, the space of mutants for **MinePump** is $\sim 10^{12}$ for mutation level 1 and $\sim 10^{34}$ for mutation level 2. Hence, to explore even the sub-space of mutants with only 1 edit ($r = 1$) we need around 300 sec for SPLALLREPAIR₁ and >400 sec (timeout) for SPLALLREPAIR₂. One way to address this problem is to use variability fault localization [5, 47], which will first identify feature and program expressions relevant for a variability bug, so that the SPL repair algorithm will apply mutations only to those expressions. This way, we will significantly reduce the space of all mutants without dropping any potentially correct mutant, and so we will improve the performance of the SPLALLREPAIR algorithm.

The runtime performance results confirm that our lifted (family-based) repair algorithm is indeed effective and especially so for large values of $|F|$ and $|K| = 2^{|F|}$. That is, the focus of lifted repair algorithm is to combat the configuration space explosion of SPLs, not their LOC or mutation space sizes. As an experiment, we took **feat-inter**, and we have gradually added optional features into it by conjoining them to the presence conditions of **#if**-s. For $|F| = 3$, SPLALLREPAIR₁ achieves speed-up of 8.3 times compared to **Brute-force**, whereas for $|F| = 4$ and $|F| = 5$ we observe speed-ups of 14.7 and 26.7 times, respectively. The key for those speed-ups is the linear growth of the running times of SPLALLREPAIR₁ with the number of features $|F|$ compared to the exponential growth of the running times of **Brute-force** with $|F|$.

Finally, the evaluation shows that for bigger values of the unwinding bound b , we obtain repairing tasks with slower runtime speed, but reporting more precise results.

6 Related Work

We divide our discussion of related work into two categories: lifted SPL analysis and program repair.

Lifted SPL analysis

Formal analysis and verification of program families have been a topic of considerable research in recent times. The challenge is to develop efficient techniques that work directly on program families, rather than on single programs. Various lifted techniques have been introduced that lift existing single-program analysis techniques to work on the level of program families. Some examples are lifted syntax checking [27, 34], lifted type checking [7, 33], lifted static analysis [6, 30, 15, 20, 55], lifted model checking [9, 16, 25], etc. There are two main lifted techniques: to develop dedicated lifted (family-based) algorithms and tools (e.g. [9, 7, 6, 20]); or to use specific simulators and variability encodings which transform program families into single programs that can be analyzed by the standard single-program analysis tools. The two approaches have different strengths and weaknesses. The advantage of the dedicated

lifted algorithms is that precise (conclusive) results are reported for every variant, but the disadvantage is that their implementation and maintenance can be labor intensive and expensive. For example, CBMC [8] is prominent (single-system) software model checker that contains many optimization algorithms, which are result of more than two decades research in advanced formal verification. Adapting and implementing all these algorithms in the context of lifted software model checking would require an enormous amount of work. Moreover, the performance of dedicated lifted algorithms still heavily depends on the size and complexity of the configuration space of the analyzed SPL.

On the other hand, the approaches based on variability encoding [30, 56] generate a family simulator that simulates the behaviour of all variants in an SPL. They re-use existing tools from single-program world, but some precision may be lost when interpreting the obtained results. The work [56] defines variability encoding on the top of TYPECHEF parser [34] for C and Java SPLs, while the work [30] defines variability encoding on the top of SUPERC parser [27] for C SPLs. The results of variability encoding have been applied to testing [35], software model checking [4], formal verification [30], and theorem proving [54] of SPLs. In this work, we pursue this line of research by presenting a lifted repair algorithm that is based on variability encoding of program families and an existing single-program mutation-based repair algorithm ALLREPAIR [50, 51].

Program repair

Automated program repair has been extensively examined in software engineering as a way to efficiently maintain software systems [28, 37, 40, 42, 45, 46, 48, 50, 51]. These works aim to repair the buggy program, so that the transformed program does not exhibit any faults. Most of them use test suits as the only specification, so the correctness of a candidate is checked by running all tests in the test suite against it. They iteratively generate a candidate from the repair search space and check its validity by testing. Some examples are GENPROG [28], RsREPAIR [48], SPR [40]. The main problem of all testing-based approaches is the generation of overfitting repairs that pass all the test cases, but they break some untested required functionality of correct programs. This happens when the test suites do not cover all the functionality of a program.

In contrast to testing-based approaches, our work belongs to the category of repair tools that use formal techniques to guide the repair process. Several techniques, such as SEMFIX [45] and ANGELIX [42], use symbolic execution to find a repair constraint and then generate a correct fix based on it. Similarly to our work, Könighofer et al. [37] also use assertions as formal specifications, but instead of mutations they use on-the-fly concolic execution (a variant of symbolic execution that uses both symbolic and concrete input values) and templates (linear expressions of program variables with unknown coefficients) as repairs. The solutions for unknown coefficients are found by SMT solving, thus discovering the repaired program. The MAPLE tool [46] utilizes a formal verification system to locate buggy expressions, which are again replaced with templates in which the unknown coefficients are determined using constraint solving. The work [36] uses a deductive synthesis framework for repairing recursive functional programs with respect to specifications expressed in the form of pre- and post-conditions.

Finally, our approach is inspired by Rothenberg and Grumberg [50, 51] that have developed the ALLREPAIR tool for automatic program repair based on code mutations. In this paper, we pursue this line of work by applying it in a new context of SPL repair, which is done by taking into account all specific characteristics of SPLs. This way, we broaden the space of programs that can be repaired.

The QLOSE tool [10] introduces a quantitative program repair algorithm that finds the “optimal” solutions by taking into account multiple quantitative objectives, such as the number of syntactic edits and semantic changes in program behaviours/executions. The work [41] proposes a semantic program repair technique that performs counterexample-guided inductive repair loop via symbolic execution. In this work, we currently find a solution with minimal number of syntactic changes to the original program family. The semantics of the program family is encoded as an SMT formula that is mutated and checked for correctness by an SMT solver. In the future, we plan to investigate some semantics-based learning techniques that will use the counterexamples returned by the SMT solver to guide the algorithm towards finding faster solutions.

Automated program repair has often been combined with fault localization. Fault localization [31, 17] is a technique for automatically generating concise error explanations in the form of locations/statements relevant for a given error that describe why the error has occurred. The works [12, 49, 51] use fault localization to narrow down the repair search space, followed by applying program repair. Firstly, fault localization suggests locations in the erroneous program that might be the cause of the error. Subsequently, the program repair attempts to change only those locations detected by the fault localization in order to eliminate the error. This way, the original program repair procedure is speeded up without incurring any precision loss. Recently, variability fault localization in buggy SPL systems has also been a subject of research [5, 44, 47]. They use spectrum-based fault localization (SBFL) metric [1] to calculate the suspiciousness scores for localizing variability bugs at the level of features [5] and statements [44, 47] based on the test information (program spectra). We can combine the variability fault localization and our variability-aware repair method to additionally prune the search space of mutants, thus improving the performances of our approach.

Program repair is also related to program sketching [52], where a program with missing parts (holes) has to be completed in such a way that a given specification is satisfied. One of the earliest and widely-known approach to solve the sketching problem is the SKETCH tool [52], which uses SAT-based counterexample-guided inductive synthesis. It iteratively performs SAT queries to find integer constants for the holes so that the resulting program is correct on all possible inputs. The works [19, 21] introduce the FAMILYSKETCHER tool that solves the sketching problem by using a lifted static analysis based on abstract interpretation. The key idea is that all possible sketch realizations represent a program family, and so the sketch search space is explored via an efficient lifted analysis of program families, which uses a specifically designed decision tree abstract domain. The FAMILYSKETCHER also generates an optimal solution to the sketching problem with respect to the number of execution steps to termination. Furthermore, the approach [18] uses abstract static analysis and logical abduction to solve the generalized program sketching problem where the missing holes can be replaced with arbitrary expressions, not only with integer constants as in the case of SKETCH and FAMILYSKETCHER tools.

7 Conclusion

In this paper, we have introduced an automated SPL repair framework using variability encoding, bounded model checking and cooperation between SAT and SMT solvers. More specifically, we utilize the CBMC bounded model checker to translate the family simulator of a program family to a program formula. By checking the satisfiability of the program formula using an SMT solver, we verify the correctness of the given program family. Then, each

formula corresponding to a buggy (feature or program) expression is replaced by a mutated patch, to create a new SMT formula that is again checked for satisfiability. To ensure that only minimally mutated programs are considered, we call a SAT solver. By experiments we have shown that our prototype tool can discover interesting patches for various buggy SPLs.

The huge space of mutants can be a bottleneck when dealing with real-world SPLs that have high sizes of LOCs and features. To overcome this problem, we can consider different techniques for pruning the search space of all possible mutations in the future. One possibility is to use variability fault localization [5, 47], which will find statements relevant for the variability bug. The formulas corresponding to all other statements will be included in S_{hard} and so no mutations will be applied to them. By mutating only statements relevant for the bug, we will significantly reduce the space of all mutants, thus speeding up the SPL repair method without any precision loss.

References

- 1 Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. Spectrum-based multiple fault localization. In *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*, pages 88–99. IEEE Computer Society, 2009. doi:[10.1109/ASE.2009.25](https://doi.org/10.1109/ASE.2009.25).
- 2 Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer, 2013. doi:[10.1007/978-3-642-37521-7](https://doi.org/10.1007/978-3-642-37521-7).
- 3 Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. Detection of feature interactions using feature-aware verification. In *26th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE 2011)*, pages 372–375, 2011. doi:[10.1109/ASE.2011.6100075](https://doi.org/10.1109/ASE.2011.6100075).
- 4 Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. Strategies for product-line verification: case studies and experiments. In *35th Inter. Conference on Software Engineering, ICSE '13*, pages 482–491, 2013. doi:[10.1109/ICSE.2013.6606594](https://doi.org/10.1109/ICSE.2013.6606594).
- 5 Aitor Arrieta, Sergio Segura, Urtzi Markiegi, Goiuria Sagardui, and Leire Etxeberria. Spectrum-based fault localization in software product lines. *Inf. Softw. Technol.*, 100:18–31, 2018. doi:[10.1016/J.INFSOF.2018.03.008](https://doi.org/10.1016/J.INFSOF.2018.03.008).
- 6 Eric Bodden, Társis Tolédo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. Splift: statically analyzing software product lines in minutes instead of years. In *ACM SIGPLAN Conference on PLDI '13*, pages 355–364, 2013.
- 7 Sheng Chen, Martin Erwig, and Eric Walkingshaw. An error-tolerant type system for variational lambda calculus. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12*, pages 29–40, 2012. doi:[10.1145/2364527.2364535](https://doi.org/10.1145/2364527.2364535).
- 8 Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Proceedings*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004. doi:[10.1007/978-3-540-24730-2_15](https://doi.org/10.1007/978-3-540-24730-2_15).
- 9 Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-François Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. *IEEE Trans. Software Eng.*, 39(8):1069–1089, 2013. doi:[10.1109/TSE.2012.86](https://doi.org/10.1109/TSE.2012.86).
- 10 Loris D’Antoni, Roopsha Samanta, and Rishabh Singh. Qlose: Program repair with quantitative objectives. In *Computer Aided Verification - 28th International Conference, CAV 2016, Proceedings, Part II*, volume 9780 of *LNCS*, pages 383–401. Springer, 2016. doi:[10.1007/978-3-319-41540-6_21](https://doi.org/10.1007/978-3-319-41540-6_21).
- 11 Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and*

12:22 Mutation-Based Lifted Repair of Software Product Lines

- Analysis of Systems, 14th International Conference, TACAS 2008. Proceedings*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008. doi:[10.1007/978-3-540-78800-3_24](https://doi.org/10.1007/978-3-540-78800-3_24).
- 12 Vidroha Debroy and W. Eric Wong. Using mutation to automatically suggest fixes for faulty programs. In *Third International Conference on Software Testing, Verification and Validation, ICST 2010*, pages 65–74. IEEE Computer Society, 2010. doi:[10.1109/ICST.2010.66](https://doi.org/10.1109/ICST.2010.66).
- 13 Aleksandar Dimovski and Danilo Gligoroski. Generating highly nonlinear boolean functions using a genetic algorithm. In *6th Int. Conference on Telecommunications in Modern Satellite, Cable and Broadcasting Service, TELSIKS 2003*, volume 2 of *IEEE*, pages 604–607, 2003. doi:[10.1109/TELSKS.2003.1246297](https://doi.org/10.1109/TELSKS.2003.1246297).
- 14 Aleksandar S. Dimovski. Symbolic game semantics for model checking program families. In *Model Checking Software - 23nd International Symposium, SPIN 2016, Proceedings*, volume 9641 of *LNCS*, pages 19–37. Springer, 2016.
- 15 Aleksandar S. Dimovski. Lifted static analysis using a binary decision diagram abstract domain. In *Proceedings of the 18th ACM SIGPLAN International Conference on GPCE 2019*, pages 102–114. ACM, 2019. doi:[10.1145/3357765.3359518](https://doi.org/10.1145/3357765.3359518).
- 16 Aleksandar S. Dimovski. Ctl^{*} family-based model checking using variability abstractions and modal transition systems. *Int. J. Softw. Tools Technol. Transf.*, 22(1):35–55, 2020. doi:[10.1007/s10009-019-00528-0](https://doi.org/10.1007/s10009-019-00528-0).
- 17 Aleksandar S. Dimovski. Error invariants for fault localization via abstract interpretation. In *Static Analysis - 30th International Symposium, SAS 2023, Proceedings*, volume 14284 of *LNCS*, pages 190–211. Springer, 2023. doi:[10.1007/978-3-031-44245-2_10](https://doi.org/10.1007/978-3-031-44245-2_10).
- 18 Aleksandar S. Dimovski. Generalized program sketching by abstract interpretation and logical abduction. In *Static Analysis - 30th International Symposium, SAS 2023, Proceedings*, volume 14284 of *LNCS*, pages 212–230. Springer, 2023. doi:[10.1007/978-3-031-44245-2_11](https://doi.org/10.1007/978-3-031-44245-2_11).
- 19 Aleksandar S. Dimovski. Quantitative program sketching using decision tree-based lifted analysis. *J. Comput. Lang.*, 75:101206, 2023. doi:[10.1016/J.COLA.2023.101206](https://doi.org/10.1016/J.COLA.2023.101206).
- 20 Aleksandar S. Dimovski and Sven Apel. Lifted static analysis of dynamic program families by abstract interpretation. In *35th European Conference on Object-Oriented Programming, ECOOP 2021*, volume 194 of *LIPICS*, pages 14:1–14:28. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:[10.4230/LIPIcs.ECOOP.2021.14](https://doi.org/10.4230/LIPIcs.ECOOP.2021.14).
- 21 Aleksandar S. Dimovski, Sven Apel, and Axel Legay. Program sketching using lifted analysis for numerical program families. In *NASA Formal Methods - 13th International Symposium, NFM 2021, Proceedings*, volume 12673 of *LNCS*, pages 95–112. Springer, 2021. doi:[10.1007/978-3-030-76384-8_7](https://doi.org/10.1007/978-3-030-76384-8_7).
- 22 Aleksandar S. Dimovski, Sven Apel, and Axel Legay. Several lifted abstract domains for static analysis of numerical program families. *Sci. Comput. Program.*, 213:102725, 2022. doi:[10.1016/J.SCICO.2021.102725](https://doi.org/10.1016/J.SCICO.2021.102725).
- 23 Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Finding suitable variability abstractions for lifted analysis. *Formal Aspects Comput.*, 31(2):231–259, 2019. doi:[10.1007/s00165-019-00479-y](https://doi.org/10.1007/s00165-019-00479-y).
- 24 Aleksandar S. Dimovski and Ranko Lazic. Compositional software verification based on game semantics and process algebra. *Int. J. Softw. Tools Technol. Transf.*, 9(1):37–51, 2007. doi:[10.1007/S10009-006-0005-Y](https://doi.org/10.1007/S10009-006-0005-Y).
- 25 Aleksandar S. Dimovski and Andrzej Wasowski. From transition systems to variability models and from lifted model checking back to UPPAAL. In *Models, Algorithms, Logics and Tools*, volume 10460 of *LNCS*, pages 249–268. Springer, 2017. doi:[10.1007/978-3-319-63121-9_13](https://doi.org/10.1007/978-3-319-63121-9_13).
- 26 Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003. doi:[10.1007/978-3-540-24605-3_37](https://doi.org/10.1007/978-3-540-24605-3_37).
- 27 Paul Gazzillo and Robert Grimm. Superc: parsing all of C by taming the preprocessor. 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 323–334. ACM, 2012. doi:10.1145/2254064.2254103.
- 28 Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012. doi:10.1109/TSE.2011.104.
 - 29 Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam A. Porter, and Gregg Rohermel. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):184–208, 2001. doi:10.1145/367008.367020.
 - 30 Alexandru F. Iosif-Lazar, Ahmad Salim Al-Sibahi, Aleksandar S. Dimovski, Juha Erik Savolainen, Krzysztof Sierszecki, and Andrzej Wasowski. Experiences from designing and validating a software modernization transformation (E). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015*, pages 597–607, 2015. doi:10.1109/ASE.2015.84.
 - 31 Manu Jose and Rupak Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*, pages 437–446. ACM, 2011. doi:10.1145/1993498.1993550.
 - 32 Christian Kästner. *Virtual Separation of Concerns: Toward Preprocessors 2.0*. PhD thesis, University of Magdeburg, Germany, May 2010.
 - 33 Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. Type checking annotation-based product lines. *ACM Trans. Softw. Eng. Methodol.*, 21(3):14, 2012.
 - 34 Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *OOPSLA '11*, pages 805–824. ACM, 2011.
 - 35 Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. Toward variability-aware testing. In *FOSD '12*, pages 1–8, 2012.
 - 36 Etienne Kneuss, Manos Koukoutos, and Viktor Kuncak. Deductive program repair. In *Computer Aided Verification - 27th International Conference, CAV 2015, Proceedings, Part II*, volume 9207 of *LNCS*, pages 217–233. Springer, 2015. doi:10.1007/978-3-319-21668-3_13.
 - 37 Robert Könighofer and Roderick Bloem. Repair with on-the-fly program analysis. In *8th International Haifa Verification Conference, HVC 2012*, volume 7857 of *LNCS*, pages 56–71. Springer, 2012. doi:10.1007/978-3-642-39611-3_11.
 - 38 Jeff Kramer, Jeff Magee, Morris Sloman, and A. Lister. Conic: An integrated approach to distributed computer control systems. *IEE Proc.*, 130(1):1–10, 1983.
 - 39 Mark H. Liffiton and Jordyn C. Maglalang. A cardinality solver: More expressive constraints for free - (poster presentation). In *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Proceedings*, volume 7317 of *LNCS*, pages 485–486. Springer, 2012. doi:10.1007/978-3-642-31612-8_47.
 - 40 Fan Long and Martin C. Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on ESEC/FSE 2015*, pages 166–178. ACM, 2015. doi:10.1145/2786805.2786811.
 - 41 Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. Semantic program repair using a reference implementation. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018*, pages 129–139. ACM, 2018. doi:10.1145/3180155.3180247.
 - 42 Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016*, pages 691–701. ACM, 2016. doi:10.1145/2884781.2884807.
 - 43 Jan Midtgård, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Systematic derivation of correct variability-aware program analyses. *Sci. Comput. Program.*, 105:145–170, 2015. doi:10.1016/j.scico.2015.04.005.

12:24 Mutation-Based Lifted Repair of Software Product Lines

- 44 Kien-Tuan Ngo, Thu-Trang Nguyen, Son Nguyen, and Hieu Dinh Vo. Variability fault localization: a benchmark. In *SPLC '21: 25th ACM International Systems and Software Product Line Conference, Leicester, United Kingdom, September 6-11, 2021, Volume A*, pages 120–125. ACM, 2021. doi:[10.1145/3461001.3473058](https://doi.org/10.1145/3461001.3473058).
- 45 Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: program repair via semantic analysis. In *35th International Conference on Software Engineering, ICSE '13*, pages 772–781. IEEE Computer Society, 2013. doi:[10.1109/ICSE.2013.6606623](https://doi.org/10.1109/ICSE.2013.6606623).
- 46 Thanh-Toan Nguyen, Quang-Trung Ta, and Wei-Ngan Chin. Automatic program repair using formal verification and expression templates. In *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Proceedings*, volume 11388 of *LNCS*, pages 70–91. Springer, 2019. doi:[10.1007/978-3-030-11245-5_4](https://doi.org/10.1007/978-3-030-11245-5_4).
- 47 Thu-Trang Nguyen, Kien-Tuan Ngo, Son Nguyen, and Hieu Dinh Vo. A variability fault localization approach for software product lines. *IEEE Trans. Software Eng.*, 48(10):4100–4118, 2022. doi:[10.1109/TSE.2021.3113859](https://doi.org/10.1109/TSE.2021.3113859).
- 48 Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *36th International Conference on Software Engineering, ICSE '14*, pages 254–265. ACM, 2014. doi:[10.1145/2568225.2568254](https://doi.org/10.1145/2568225.2568254).
- 49 Urmas Repinski, Hanno Hantson, Maksim Jenihhin, Jaan Raik, Raimund Ubar, Giuseppe Di Guglielmo, Graziano Pravadelli, and Franco Fummi. Combining dynamic slicing and mutation operators for ESL correction. In *17th IEEE European Test Symposium, ETS 2012*, pages 1–6. IEEE Computer Society, 2012. doi:[10.1109/ETS.2012.6233020](https://doi.org/10.1109/ETS.2012.6233020).
- 50 Bat-Chen Rothenberg and Orna Grumberg. Sound and complete mutation-based program repair. In *FM 2016: Formal Methods - 21st International Symposium, Proceedings*, volume 9995 of *LNCS*, pages 593–611, 2016. doi:[10.1007/978-3-319-48989-6_36](https://doi.org/10.1007/978-3-319-48989-6_36).
- 51 Bat-Chen Rothenberg and Orna Grumberg. Must fault localization for program repair. In *Computer Aided Verification - 32nd International Conference, CAV 2020, Proceedings, Part II*, volume 12225 of *LNCS*, pages 658–680. Springer, 2020. doi:[10.1007/978-3-030-53291-8_33](https://doi.org/10.1007/978-3-030-53291-8_33).
- 52 Armando Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013. doi:[10.1007/s10009-012-0249-7](https://doi.org/10.1007/s10009-012-0249-7).
- 53 Shin Hwei Tan, Jooyong Yi, Yulis, Sergey Mechtaev, and Abhik Roychoudhury. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017 - Companion Volume*, pages 180–182. IEEE Computer Society, 2017. doi:[10.1109/ICSE-C.2017.76](https://doi.org/10.1109/ICSE-C.2017.76).
- 54 Thomas Thüm, Ina Schaefer, Martin Hentschel, and Sven Apel. Family-based deductive verification of software product lines. In *Generative Programming and Component Engineering, GPCE'12*, pages 11–20. ACM, 2012. doi:[10.1145/2371401.2371404](https://doi.org/10.1145/2371401.2371404).
- 55 Alexander von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven Apel. Variability-aware static analysis at scale: An empirical study. *ACM Trans. Softw. Eng. Methodol.*, 27(4):18:1–18:33, 2018. doi:[10.1145/3280986](https://doi.org/10.1145/3280986).
- 56 Alexander von Rhein, Thomas Thüm, Ina Schaefer, Jörg Liebig, and Sven Apel. Variability encoding: From compile-time to load-time variability. *J. Log. Algebraic Methods Program.*, 85(1):125–145, 2016. doi:[10.1016/j.jlamp.2015.06.007](https://doi.org/10.1016/j.jlamp.2015.06.007).

Pure Methods for roDOT

Vlastimil Dort 

Charles University, Prague, Czech Republic

Yufeng Li 

University of Cambridge, UK

Ondřej Lhoták 

University of Waterloo, Canada

Pavel Parízek 

Charles University, Prague, Czech Republic

Abstract

Object-oriented programming languages typically allow mutation of objects, but pure methods are common too. There is great interest in recognizing which methods are pure, because it eases analysis of program behavior and allows modifying the program without changing its behavior. The roDOT calculus is a formal calculus extending DOT with reference mutability. In this paper, we explore purity conditions in roDOT and pose a SEF guarantee, by which the type system guarantees that methods of certain types are side-effect free. We use the idea from ReIm to detect pure methods by argument types. Applying this idea to roDOT required just a few changes to the type system, but necessitated re-working a significant part of the soundness proof. In addition, we state a transformation guarantee, which states that in a roDOT program, calls to SEF methods can be safely reordered without changing the outcome of the program. We proved type soundness of the updated roDOT calculus, using multiple layers of typing judgments. We proved the SEF guarantee by applying the Immutability guarantee, and the transformation guarantee by applying the SEF guarantee within a framework for reasoning about safe transformations of roDOT programs. All proofs are mechanized in Coq.

2012 ACM Subject Classification Software and its engineering → Formal language definitions; Software and its engineering → Object oriented languages

Keywords and phrases type systems, DOT calculus, pure methods

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.13

Related Version *Extended Version including Appendix:* https://d3s.mff.cuni.cz/files/publications/dort_pure_report_2024.pdf [13]

Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact):* <https://doi.org/10.4230/DARTS.10.2.6>

Funding This work was supported by the Czech Science Foundation project 23-06506S, and by the Czech Ministry of Education, Youth and Sports project LL2325 of the ERC.CZ programme. This research was also supported by the Natural Sciences and Engineering Research Council of Canada.

1 Introduction

A feature common to many object-oriented programming languages is that execution of a method can have important side effects such as creating new objects on the heap or modifying (mutating) existing objects. For example, a setter method modifies a field of the receiving object. Such effects are also the reason why, in general, execution of a method cannot be treated as evaluation of a function in a mathematical sense, because every call of a method with possible side effects can produce different results.

 © Vlastimil Dort, Yufeng Li, Ondřej Lhoták, and Pavel Parízek;
licensed under Creative Commons License CC-BY 4.0
38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 13; pp. 13:1–13:29



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

13:2 Pure Methods for roDOT

That being said, many methods in object-oriented programs are actually designed as side-effect-free and meant to work like pure mathematical functions, producing the same result on each invocation. An example of such methods are getters, or generally, computations based solely on the arguments passed into the method. Creating methods without side effects is also often considered to be a good practice, because it reduces hidden dependencies, and these methods can be used more freely without the fear of unwanted interaction of their effects. For example, the program code fragment `val x = computeX() ; val y = computeY()`, which involves two side-effect-free methods, can be transformed to `val y = computeY() ; val x = computeX()` by swapping the order of method calls without any observable change in the program behavior and semantics. Writing side-effect-free methods also enables a greater degree of parallelization (concurrency) and, in general, makes it easier to understand the program behavior. Therefore, the issue of purity is relevant to most mainstream object-oriented programming languages, such as Java, C++, C# and Scala.

However, in common programming languages, pure functions and methods with effects are typically unified under a single concept of a method, and there is no way to express, check and make use of method purity at the language level. The idea that a method is pure can be expressed using an annotation (see, e.g., Checker Framework [14, 10] and Code Contracts [15]), but one must look into the documentation of such an annotation for the exact meaning of purity, and there may be limited possibilities of checking automatically whether the annotation is applied properly.

In the context of Java, ReIm [19] introduced annotations with a formal meaning, which give rise to a type system that allows to recognize side-effect-free methods using the types of their parameters – if all parameters of a method, including the receiver, have read-only types, the method cannot get hold of a writeable reference to an existing object, so it is necessarily side-effect free. The advantage of this general approach, based on the usage of static type systems for reasoning about purity and side-effect freedom, is the possibility to prove soundness and consistency of such annotations.

Scala favours a functional programming style, so Scala programs are likely to contain more methods (than Java programs) that can be identified as side-effect-free. Our main objective is to design a type system that guarantees side effect freedom for Scala methods and supports advanced language features present in Scala.

Previous formalization efforts for Scala resulted in the Dependent Object Types (DOT) calculus [2], which captures the essence of Scala’s type system. However, the original DOT calculus does not model mutation of objects, so purity cannot be addressed there, but some variants that do allow mutation have been developed. roDOT [12] is an existing core calculus for Scala with reference mutability. It has mutable fields and a type system feature to distinguish read-only and mutable references. An important rationale behind the design of roDOT, when compared to other possible approaches, is to use existing features of Scala, including its rich type system, as much as possible rather than introducing new forms of types only for reference mutability, to ease adoption of such a type system into the Scala language. In particular, roDOT expresses the mutability of a reference using a specially designated type member in the type of that reference. The type system of roDOT also provides an *immutability guarantee*: an object can only be mutated if there is a path of mutable references to it from the code being executed.

In this paper, we extend the core roDOT calculus from [12] with the concept of side-effect-free methods. Before going into details, we want to emphasize that it was not possible to simply adapt ReIm [19] from Java, because of several challenges specific to Scala and roDOT that we discuss below. However, we use the idea proposed by the authors of ReIm that side-effect-free methods are recognized based on the types of their parameters.

The general concept of purity is, in addition to (1) side-effect-freedom, sometimes understood to comprise more properties: (2) determinism – returning the same value for the same arguments [14], and (3) termination. In this paper, we focus only on the side-effect-free (SEF) property. We will just mention that in regards to determinism, mutable DOT calculi including roDOT have semantics that is deterministic except for instantiation of objects.

Within the context of roDOT, we look at the SEF property from three different perspectives – what a SEF method does, how it can be recognized using the type system, and how it can be used in programs. We define SEF methods in roDOT as those that do not modify any objects that existed on the heap before the method was called.

As the main result of this paper, we prove the *side-effect-free guarantee* (SEF guarantee), which says that methods with read-only parameters do not modify existing objects.

One important related challenge is that in order to state and prove the SEF guarantee, we needed a way to test whether a given type is read-only. As we will explain, this is not possible in the existing (original) roDOT type system from [12]. Therefore, one of our contributions is an extension of roDOT that makes it possible to recognize read-only types.

Another challenge was defining the SEF guarantee formally and proving it in a calculus that supports a mutable heap (like roDOT). The roDOT operational semantics says that fresh heap addresses are chosen during method execution. Therefore, after calling a SEF method, these heap addresses can be different, yet the heap still has the same overall structure. We formally define a concept of similarity of heaps in roDOT to describe this relation. We prove the SEF guarantee by simulating the execution of a SEF method with a similar execution, where writeable references are removed, and by applying roDOT's immutability guarantee.

Finally, as a corollary of the SEF guarantee, we state and prove a guarantee of safety of a particular code transformation. The transformation guarantee states that swapping two calls to SEF methods anywhere in a program does not affect the result of its execution.

Formalizing safe program transformations has to deal with specific issues, such as mixing of program code and values together on the program heap, or the heap similarity mentioned above. In order to deal with these issues, we design a general framework for reasoning about safe transformations in roDOT. The framework provides a general way to define program transformations, defines what properties a safe program transformation must have, and provides a general theorem about lifting the safety of transformation from execution of a small piece of code to execution of the whole program. Within this framework, we define the specific transformation of swapping two calls of SEF methods. We prove the transformation guarantee using the SEF guarantee and the lifting theorem.

We mechanized all of our formal results, in particular the soundness proof of the extended roDOT calculus and the SEF guarantee, in Coq to enable future formal reasoning to build on them. Note that the soundness of the original roDOT calculus was proved by hand in [12]. We have made our formalization in Coq public as an artifact for this paper.

1.1 Contribution

In summary, the main contributions of this paper are the following:

- a modification of the original roDOT calculus that makes it possible to test whether a type is read-only, which is necessary to state and prove the SEF guarantee;
- a formal definition of side-effect-free methods in the context of roDOT, statement and proof of the SEF guarantee;

- a general framework for defining transformations of roDOT programs and proving that some of them are safe in that they do not change the result of program execution, statement and proof of a transformation guarantee, which states that re-ordering calls to SEF methods is safe in that sense;
- the first mechanization of roDOT and its immutability guarantee, with addition of the SEF and transformation guarantees, and all the proofs in Coq – provided as an artifact.

1.2 Outline

The paper is organized as follows. Section 2 gives an overview of the roDOT calculus, which has been mechanized in Coq and within which we define the SEF condition. Section 3 discusses the definition of pure and SEF methods, looking at several variants. It defines the SEF guarantee and identifies necessary changes to the roDOT type system in order for the guarantee to work. In Section 4 we describe the changes to the calculus in more detail, and discuss a new proof of type soundness of the calculus. In Section 5, we describe how we proved the SEF guarantee, and in Section 6 we define and prove the transformation guarantee within a framework for safe transformations. An appendix containing full definitions and more detailed discussion is available in the extended version of this paper [13].

2 Background – The roDOT calculus

In this section, we present the summary of the roDOT calculus [12], which we use as the baseline for this work. The DOT calculus [2, 33, 30] is a formal calculus, designed to formalize the essence of the types of the Scala programming language. In the basic versions of the DOT calculus, objects have read-only fields (so the objects are immutable), but there are also several versions that allow changing values of the fields of objects (mutation).

The roDOT calculus [12] evolved from DOT with mutable fields. The goal was to extend DOT with the ability to control mutation of objects using the type system, while using the existing features of the DOT calculus, dependent types.

In roDOT, write access to a field is controlled by a reference mutability permission. It is based on an idea of a reference capability represented by a special type member M . A reference can only be used to mutate an object if the type of the reference includes this capability, in the form of a type member declaration $\{M : \perp\ldots\perp\}$. Thanks to that, we can refer to the mutability of a variable x using type selection $x.M$.

Without this capability, the field can only be read, but with it, the field can also be written to. The permission applies transitively, in the sense that reading from a read-only reference always produces read-only references.

2.1 Syntax and typing

The syntax of terms and types in roDOT is in Figure 1. It uses the A-normal form [36] of terms from DOT. To avoid ambiguity, if a variable is used in the position of a term, it is marked as vx . Unlike other versions, the roDOT calculus does not have λ values, but methods are a kind of object member (and cannot be reassigned), so there is a more explicit relationship of a method, the containing object and the reference used to call the method. Objects are represented by the $\nu(s : R)d$ constructor, appearing as literals in the programs and as items on the heap (R is the type of the object and d is a list of member definitions).

When typing the program or a part of it, free variables are assigned a type in a typing context Γ . There are several kinds of variables. *Abstract variables* are variables bound in terms such as let-in terms and method definitions. When the program executes, objects are

Variable	Type
$x ::= z, s, r$	
$ y \mid w$	location, reference
$t ::=$	Term
$ vx \mid x_1.m\,x_2$	variable, method call
$ \text{let } z = t_1 \text{ in } t_2$	let
$ \text{let } z = \nu(s : T)d \text{ in } t$	let-literal
$ x.a \mid x_1.a := x_2$	read, write
$d ::= d_1 \wedge d_2$	Definition
$ \{a = x\} \mid \{A(r) = T\}$	field, type
$ \{m(z, r) = t\}$	method
$\rho ::= \cdot \mid \rho, w \rightarrow y$	Environment
$T ::=$	
$ \top \mid \perp \mid \mathsf{N}$	top, bottom, read-only \perp
$ \mu(s : T) \mid x_1.B(x_2)$	recursive, selection
$ \{a : T_1..T_2\} \mid \{B(r) : T_1..T_2\}$	field, type decl.
$ \{m(z : T_1, r : T_3) : T_2\}$	method
$ T_1 \wedge T_2 \mid T_1 \vee T_2$	intersection, union
$B ::=$	Type member name
$ A \mid \mathsf{M}$	ordinary, mutability
$\sigma ::= \cdot \mid \text{let } z = \square \text{ in } t :: \sigma$	Stack
$\Sigma ::= \cdot \mid \Sigma, y \rightarrow d$	Heap
$c ::= \langle t; \sigma; \rho; \Sigma \rangle$	Configuration

■ **Figure 1** roDOT syntax.

$$\begin{array}{c}
 \frac{\Gamma; \rho \vdash x_1 : \{m(z : T_1, r : T_3) : T_2\}}{\Gamma; \rho \vdash x_2 : T_1 \quad \Gamma \mathbf{vis} x_2} \quad \frac{\Gamma \mathbf{vis} x_1}{\Gamma; \rho \vdash x_1 : [x_2/z]T_3 \quad \Gamma \mathbf{vis} x_1} \\
 \frac{T_3 \mathbf{indep} z}{\Gamma; \rho \vdash x_1.m\,x_2 : [x_1/r][x_2/z]T_2} \quad (\text{TT-Call}) \quad \frac{\Gamma; \rho \vdash x_1 : T_1 \quad \Gamma \mathbf{vis} x_1}{\Gamma; \rho \vdash x : \{a : T_1..T_2\} \quad \Gamma \mathbf{vis} x} \\
 \frac{\Gamma; \rho \vdash x : \{a : T_1..T_2\} \quad \Gamma \mathbf{vis} x}{\Gamma; \rho \vdash T_2 \mathbf{ro} T_3 \quad \Gamma; \rho \vdash T_2 \mathbf{mu}(r) T_4} \quad (\text{TT-Read}) \quad \frac{\Gamma; \rho \vdash x : \{\mathsf{M}(r) : \perp..\perp\}}{\Gamma; \rho \vdash x.a := x_1 : T_2} \quad (\text{TT-Write}) \\
 \frac{\Gamma; \rho \vdash T_1 <: T_3 \quad \Gamma; \rho \vdash T_2 <: T_3}{\Gamma; \rho \vdash T_1 \vee T_2 <: T_3} \quad (\text{ST-Or}) \\
 \frac{\Gamma; \rho \vdash T_1 \wedge (T_2 \vee T_3) <: (T_1 \wedge T_2) \vee (T_1 \wedge T_3)}{\Gamma; \rho \vdash T_1 \wedge (T_2 \vee T_3) <: (T_1 \wedge T_2) \vee (T_1 \wedge T_3)} \quad (\text{ST-Dist}) \\
 \frac{\Gamma; \rho \vdash T_3 <: T_1 \quad \Gamma, z : T_3, r : T_6; \rho \vdash T_2 <: T_4}{\Gamma, z : T_3; \rho \vdash T_6 <: T_5 \quad T_6 \mathbf{indep} z \Rightarrow T_5 \mathbf{indep} z} \\
 \frac{\Gamma; \rho \vdash \{m(z : T_1, r : T_5) : T_2\} <: \{m(z : T_3, r : T_6) : T_4\}}{\Gamma; \rho \vdash \{m(z : T_1, r : T_5) : T_2\} <: \{m(z : T_3, r : T_6) : T_4\}} \quad (\text{ST-Met})
 \end{array}$$

■ **Figure 2** Selected rules for typing and reduction in roDOT.

created on the heap, and variables referring to concrete objects on the heap are substituted in place of the abstract variables. Each object on the heap has a unique location y and one or more references w . In an object on the heap, the values of fields are locations of other objects. In terms, only references may appear. The kind of the variable has no effect on execution or typing. In roDOT, references are a separate concept from locations in order to allow references to the same object to have different types (specifically, different mutabilities). While the run-time stack and focus of execution work with references that have their own mutabilities, the heap only works with locations, and mutability is determined by field types.

The types form a lattice, with the top, bottom, union and intersection types. Objects can contain multiple members – fields, methods and type members. Types of objects are formed by intersection of individual declaration types for each member.

The type members $\{A : T..T\}$ specify lower and upper bounds, and they introduce a new dependent type $x.A$ that has a subtyping relationship with those bounds. This is relevant because roDOT uses a type member for mutability. The ability to create dependent types in this manner is the defining feature of the DOT calculus.

The declarations of an object’s members are wrapped in a recursive type, so several declarations in one object type can reference each other, using a member type selection $s.A$ involving the self-reference s . An example of a type of an object without mutability is $\mu(s : \{A : T..T\} \wedge \{a : s.A..s.A\} \wedge \{m(r : T, z : T) : T\})$. An object of this type has a type member A with bounds T , a field a with a self-referential type $s.A$, and a method m .

In roDOT, dependent types are also used to express the mutability of a reference, by selecting the special type member M . When accessing an object through a reference which does not have this capability, for example $\{a : T..T\}$, the field can only be read. With it, for example $\{a : T..T\} \wedge \{M : \perp..\perp\}$, the field can also be written to.

In the declaration of the type member M , the lower bound is always \perp , and the upper bound determines the mutability. If the upper bound is also \perp , it means the reference is mutable. Otherwise, it is read-only. This way, mutable references are subtypes of read-only references, so a mutable reference can be used anywhere a read-only reference is expected, but not vice versa. We will use M_T as a shorthand for the type member declaration $\{M : \perp..T\}$, or just M when the bound is not important. The mutability of a reference applies to the whole object – a mutable reference allows writing to all fields.

An example of a type of an object with a type member A , a field a , method m and a mutability declaration is $\mu(s : \{A : T..T\} \wedge \{a : T..T\} \wedge \{m(r : T, z : T) : T\}) \wedge \{M : \perp..\perp\}$.

A declaration of a method allows specifying a type of the receiving reference $r : T$, which can be more precise than the type of the recursive self parameter s in the defining object. This allows the type of the method to require that the receiver be writeable, or allow it to be read-only. It is similar to the ability to annotate the type of `this` parameter in Java, used by the Checker Framework [18, 9]. For this reason, every method in roDOT has two parameters: a normal parameter z and the receiver r , which is a reference to the object containing the method, like `this` in Scala. In roDOT, the type of r can be dependent on z . The parameter r is special in how it gets its type, but in terms of semantics, behaves the same as z .

Several rules in roDOT need a read-only version of a type. For that, there are two type-level operations: $T \mathbf{ro} U$ means that U is a readonly version of T , $T \mathbf{mu} U$ means that U is a mutability bound of type T (rules are shown in Figure 11 in the appendix). A special type N is defined to be the read-only version of the least type in the subtyping lattice, \perp .

The typing rules (selected in Figure 2, full set in Figures 8 to 12 in the appendix) describe correctly formed programs. In addition to the typing context Γ , which assigns types to variables, the left side of the typing judgment includes an environment ρ that connects references in the terms to locations of objects on the heap.

The write term, typed by TT-Write, is guarded by a check of the mutability permission on the receiving reference. The premise $\Gamma; \rho \vdash x : \{M : \perp..\perp\}$ ensures that only mutable references can be used for writing.

Reading a field, typed by TT-Read, is always possible, but the type of the result is changed to read-only if the source reference is read-only. This type operation is called viewpoint adaptation, and ensures that read-onlyness is transitive, which is required for the immutability guarantee of roDOT and for our SEF guarantee. This is achieved by taking a read-only version of the field’s type, and adding a mutability that is a union of the mutabilities of the source reference and of the field type. For example, if a reference w has type $\{a : T_1..\mu(s : \dots) \wedge M_U\}$, then the term $w.a$ has type $\mu(s : \dots) \wedge M_{w.a \vee U}$.

With the **vis** judgment (Figure 9 in the appendix), roDOT hides captured variables in methods – to access a value from outside, it must be stored in a field of the containing object, so viewpoint adaptation applies to it.

Variables appearing in terms and definitions have types given by the typing and subtyping rules in Figures 9 and 10 in the appendix. Selected rules are shown in Figure 2: ST-Met is a subtyping rule for method declarations. The part highlighted in grey is not part of roDOT, but our modification, which we will describe in Section 4.2. Rules ST-Or and ST-Dist are examples of subtyping rules for union types, which are relevant in Section 4.1.

2.2 Semantics

The operational semantics of roDOT is defined as a small step semantics, with machine configurations (Figure 1) consisting of a term in the focus of execution t , a stack σ , a heap Σ and an environment ρ . The environment ρ maps references to locations and the heap Σ maps locations to objects. The stack σ is used to evaluate terms of the form $\text{let } z = t_1 \text{ in } t_2$. The stack is a list of frames of the form $\text{let } z = \square \text{ in } t_2$, where \square represents t_1 while it is being evaluated in the focus of execution. When t_1 is evaluated to a value, that value is substituted for the square in the top frame of the stack, and the t_2 from that frame then becomes the new focus of execution.

Execution starts with the program, an empty stack, empty heap and an empty environment, and proceeds by steps defined in Figure 13 in the appendix, until it reaches an answer configuration, which has an empty stack and the focus of execution is a single variable. During execution, new items are added to the heap and the environment (there is no garbage collection). Calling a method copies its body to the focus of execution, while the receiver and argument are substituted.

The semantics is generally deterministic – there is no way to express a nondeterministic choice. However, there is one source of non-determinism: locations of objects on the heap. Allocating objects must be regarded as a non-deterministic operation because even if the new objects are initially equal, they may take on different values due to subsequent mutation.

2.3 Properties

The roDOT calculus has the type soundness property (Theorem 1, Theorem 7 in [12]) – a term that has a type in an empty context can be executed and either reduces to an answer, or executes indefinitely. DOT and roDOT do not include explicit checks for error conditions, but trying to access (read, write or call) a non-existing member of an object is an error. In such a case, a reduction step is not defined and the execution “gets stuck”. The soundness theorem guarantees this does not happen for typed programs.

► Theorem 1 (Type Soundness).

If $\vdash t_0 : T$, $\left| \begin{array}{l} \text{The initial term } t_0 \text{ is well typed,} \\ \text{then either } \exists w, j, \Sigma, \rho: \langle t_0; \cdot; \cdot; \cdot \rangle \xrightarrow{j} \langle vw; \cdot; \rho; \Sigma \rangle, \\ \text{or } \forall j: \exists t_j, \sigma_j, \Sigma_j, \rho_j: \langle t_0; \cdot; \cdot; \cdot \rangle \xrightarrow{j} \langle t_j; \sigma_j; \rho_j; \Sigma_j \rangle. \end{array} \right.$
then execution terminates in j steps with answer w ,
or continues indefinitely.

Type soundness and other properties are based on the fact that during execution, the type of the configuration is preserved. Rules for typing a machine configuration are in Figure 14 in the appendix. As the program executes and new objects are added to the heap, new locations and reference variables are used to refer to the objects. To give the configurations a type, these variables are added to the typing context. Their type is the type of the object, and has a fixed form – it is a recursive type containing declarations of all the object’s members, intersected with a declaration of mutability. A typing context that only contains types of this form is called an *inert context*. Under an inert context, stronger claims can be made about types of variables [30], and it plays an important role in the proof of soundness.

$$\begin{array}{c}
 \frac{\Gamma \vdash \langle t; \sigma; \rho; \Sigma \rangle \text{ mreach } y_1}{y_1 \rightarrow \dots_1 \{a = y_2\} \dots_2 \in \Sigma} \\
 \frac{\Gamma; \rho \vdash y_1 : \{a : \perp \dots \{M(r) : \perp \dots \perp\}\}}{\Gamma \vdash \langle t; \sigma; \rho; \Sigma \rangle \text{ mreach } y_2} \text{ (Rea-Fld)} \quad \frac{t \text{ tfree } w \vee \sigma \text{ tfree } w}{w \rightarrow y \in \rho} \\
 \frac{\Gamma; \rho \vdash w : \{M(r) : \perp \dots \perp\}}{\Gamma \vdash \langle t; \sigma; \rho; \Sigma \rangle \text{ mreach } y} \text{ (Rea-Term)}
 \end{array}$$

Figure 3 roDOT mutable reachable references.

The essential property of roDOT is the immutability guarantee (Theorem 2, Theorem 9 in [12]): in order for an object to be mutated, a writeable reference to it must exist, or it must be possible to reach it by a path of writeable fields, starting from a writeable reference – the object must be mutably reachable, defined formally in Figure 3.

► **Theorem 2 (Immutability Guarantee).**

If $y \rightarrow d \in \Sigma_1$ and $\Gamma \vdash \langle t_1; \sigma_1; \rho_1; \Sigma_1 \rangle : T$,
and $\langle t_1; \sigma_1; \rho_1; \Sigma_1 \rangle \xrightarrow{k} \langle t_2; \sigma_2; \rho_2; \Sigma_2 \rangle$,
then either $y \rightarrow d \in \Sigma_2$,
or $\Gamma \vdash \langle t_1; \sigma_1; \rho_1; \Sigma_1 \rangle \text{ mreach } y$.

For an object at some point during well-typed execution,
at any later point,
either the object does not change,
or it was reachable by mutable references.

3 Method Purity for roDOT

Here we informally define the meaning of side-effect freedom in roDOT, and informally state the main results of this paper: the SEF guarantee and the transformation guarantee.

We structure our work around an observation that (in any programming language or calculus), we can look at side-effect freedom from different perspectives:

1. (Static) Recognize which methods are SEF statically at compile time, using types.
2. (Runtime) Define what events can (or cannot) happen when a SEF method is executed.
3. (Usage) Differentiate SEF methods from general methods based on how they can be safely used in programs.

For each of these perspectives, we will state a *SEF condition*, each giving a different definition of SEF methods in roDOT. First we do it informally in this section, and then formalize the definitions in the following sections. The guarantees then form connections between different SEF conditions.

3.1 Runtime SEF condition

Saying that a method is side-effect-free is informally understood as saying that the execution of the method will not perform any actions that are considered to be side effects. This view corresponds to the second perspective on our list.

This perspective is most directly related to the semantics. In roDOT, this means looking at the small step semantics, defining the beginning and end of execution of a method, and defining the SEF condition in terms of the state of execution or the steps performed between the beginning and the end. When looking at the effects caused by method execution, the only relevant part of the machine configuration is the heap (the focus of execution is the part being evaluated, the stack cannot be changed, and the mapping from references to locations is only relevant for typing). The heap can only be modified by two kinds of execution steps: instantiation of an object and writing a value to a field of an object on the heap.

The condition of side-effect-freedom can be stated in multiple versions of varying strength. In the strictest sense, we could say that a SEF method cannot have any effect on the heap at all, meaning no instantiations and no writes. That would, however, be overly restrictive, as object instantiation is one of the basic operations in object-oriented programming. It is therefore usually (such as in [34, 38, 19, 14]) allowed that a SEF method can instantiate new objects, and also write to the fields of those newly instantiated objects. In turn, the only forbidden action is writing to fields of previously existing objects.

Another choice in the definition is when the change to the heap is detected, which leads to different answers to questions such as: (a) Is it allowed to write to a field of an existing object, if the value written is the same as the current value so the object does not actually change? (b) Is it allowed to write to a field of an existing object, if the field is restored to the previous value before the end of the method execution? We choose to allow (a) but not (b), so our definition observes the state of the heap at every moment during the execution of the method. Allowing (b) would lead to a weaker condition, which would check the state of the heap only at the end of the method call. Forbidding (a) would lead to a stronger condition, defined in terms of allowed steps of execution rather than in terms of the state.

► **Informal statement of Definition 15** (Run-time SEF condition, in Section 5.1). An execution of a method is side-effect free, when at every step of execution until returning from the method, the heap contains all the objects from the start of execution in an unchanged state.

3.2 Static SEF condition

The static perspective (the first in our list) is useful because it provides a way to check that a method is SEF by looking at the code. We must, however, accept that statically, it will not be possible to recognize all methods that are pure from the second (and third) perspective.

In ReIm [19], SEF methods are recognized by the mutability of the parameters. roDOT uses the same notion of transitive read-only references, therefore it should be possible to use an analogous condition in roDOT.

► **Informal statement of Definition 11** (static SEF condition). A method has a SEF type, if both its parameter and its receiver parameter have read-only types.

This condition will be formally defined in Section 4.1. Example 3 and Example 4 illustrate its ability to recognize SEF methods.

► **Example 3.** A getter defined as $\{m_{get}(r, z) = z.a\}$ can be typed with $\{m_{get}(r : \top, z : \{a : \top..\top\} : \top)\}$. Both \top and $\{a : \top..\top\}$ are read-only types, and therefore the getter is SEF.

► **Example 4.** The method m_{sef} defined by $\{m_{sef}(r, z_a) = (\text{let } x = \nu(r_o : R_o) \dots \text{in } z_a.m_a x)\}$ calls a method of its argument, passing a newly allocated object to it. This method has type $\{m_{sef}(r : \top, z_a : T_z) : \top\}$, where $T_z = \{m_a(r : \top, z : \mu(r_o : R_o) \wedge \{\mathbf{M} : \perp..\perp\}) : \top\}$. By Definition 11, m_{sef} is SEF, because it has read-only parameters, even though it calls m_a , which may mutate the heap.

Example 5 shows how viewpoint adaptation transitively ensures that read-only parameters cannot be used to modify existing objects. Example 6 shows how a dependent type can change whether the method is SEF or possibly not.

► **Example 5.** The method defined by $\{m_{va}(r, z) = (\text{let } x = z.a \text{ in } x.b := r)\}$ mutates an object stored in a field of the argument z , and therefore is not SEF. This method cannot be typed with a read-only type for the parameter z , because even if the field a has a mutable type, by viewpoint adaptation of fields in roDOT, the variable x would also have a read-only type, so the subsequent write would not be allowed.

► **Example 6.** A method with a type $\{m_{dep}(r : \top, z_a : \{a : \top..\top\} \wedge x.A) : \top\}$ has a parameter with a type dependent on the variable x , which can decide the mutability. This method is recognized as SEF only in contexts where $N <: x.A$. When $x.A <: M_\perp$, then the method can (indirectly) mutate the argument.

3.3 SEF guarantee

For the **SEF guarantee** (Theorem 16), we want to be able to claim that a method is SEF based on the type of the method declaration. The SEF guarantee makes the connection from the first to the second perspective.

► **Informal statement of Theorem 16** (SEF guarantee, in Section 5.2). Let c_1 be a well-typed machine configuration just prior to executing a method call step $w_1.mw_2$. If, by typing of the receiving reference w_1 , the method m has a SEF type, then the execution of the method will be side-effect free.

3.4 Using pure methods in roDOT

Finally, the third perspective shows why SEF methods are useful. It is, however, a view from outside of the method, and does not tell us how to construct a SEF method or check it.

The practical use of a type system with SEF methods comes when it allows us to look at the code, and based on what we see (from the first perspective) gives us a guarantee about its behavior (second perspective) and how it can be used (the third perspective). An example of this is allowing safe transformations of the program, which can be applied at coding time using IDE-provided code transformations, or at compile time as optimizations. For example, calls to SEF methods can be safely reordered.

To keep the problem simple, we will look at one particular case of such reordering: swapping two calls to SEF methods. With SEF methods, the code $x1.m1(); x2.m2()$ is equivalent $x2.m2(); x1.m1()$.

► **Informal definition** (Call-swapping transformation of programs). A program t_1 is transformed into t_2 by SEF call-swapping, when the programs are the same except in one place, where t_1 calls two methods in succession, but t_2 calls them in the opposite order. Furthermore, within the contexts of typing these method calls, both methods have the same read-only types, and allow both programs to be typed in the same manner.

► **Example 7.** A chain of calls $\text{let } x_1 = x_{o1}.m_1x_{a1} \text{ in let } x_2 = x_{o2}.m_2x_{a2} \text{ in } t$, can be transformed by call swapping into $\text{let } x_2 = x_{o2}.m_2x_{a2} \text{ in let } x_1 = x_{o1}.m_1x_{a1} \text{ in } t$.

The static condition from the first perspective is already a part of the definition of the transformation. The transformation guarantee then states that this transformation is safe – it does not change the behavior of the program. By that, the guarantee connects the static condition (first perspective) with the call-swapping transformation (third perspective). We use the run-time condition (second perspective) as a connecting step between them in the proof of this guarantee.

► **Informal statement of Theorem 27.** The call-swapping transformation is safe, in the sense that if for any programs t_1 and t_2 related by this transformation, provided that t_1 terminates with an answer c_1 , then t_2 also terminates with an answer c_2 , which is the same as c_1 , except for certain unavoidable differences in variable names and in method bodies.

The formal definition of the transformation, formal statement of the transformation guarantee and an outline of its proof are provided in Section 6.

4 Recognizing SEF methods by type in modified roDOT

In this section, we formalize the static SEF condition in roDOT given informally in Section 3.2. Although the notion of read-only types, used by this condition, was already defined in roDOT, we identify issues with that definition in regards to this new use.

We fix them by updating the calculus with small changes, which comprise adding one new subtyping rule and one type splitting rule, and one restriction added to the method subtyping rule. The updated calculus is neither a subset nor a superset of the original, so it is necessary to update the proof of soundness and the immutability guarantee, which were proven by hand for the original roDOT [12]. The soundness proof followed the scheme from [30] and uses an auxiliary definition of invertible typing, which allows doing proofs by induction on the typing of variables. This is possible thanks to eliminating possible cycles in the derivation, by forcing the derivation to follow the syntactic structure of the target type.

One of the new subtyping rules, however, breaks this soundness proof, because it introduces new possibilities to derive types in cycles, which cannot be repaired by simply handling additional cases in the original proof. In the presence of cycles, we cannot use the straightforward inductive hypothesis to prove properties necessary for type safety, because a derivation for typing a variable can involve derivations of arbitrarily complex types.

We implemented a new proof based on a different auxiliary typing definition, which avoids cycles by forcing the derivation to arrive at the target type by adding type constructors in a fixed order (for example, all unions in the type are handled before intersections). Compared to the original invertible typing, which was single typing judgment with many rules, the new approach leads to a definition in several layers, where each layer has a small number of typing rules. We call this set of judgments *layered typing*. In layered typing, we re-prove important properties of invertible typing, so that the new definition fits into the rest of the existing soundness proof, and also prove new properties required for the SEF guarantee.

The rest of this section is structured as follows: in Section 4.1, we formalize static SEF condition, and discuss the meaning of read-only types in roDOT. In Section 4.2, we propose small changes to the roDOT calculus to make definitions work for the SEF guarantee. We give a short overview of the structure of the original soundness proof for roDOT, and show how this proof breaks with the new changes. In Section 4.3, we describe the new layered typing that replaces invertible typing in the updated proof and show its important properties.

4.1 Static SEF condition in roDOT

In the **SEF guarantee** (Theorem 16), we claim that a method is SEF based on the type of the method declaration. Our SEF guarantee follows the approach of ReIm [19] and requires the parameters to have read-only types.

4.1.1 Read-only types in roDOT

The check whether a type is read-only was also present in roDOT, but it had a limited purpose – to ensure that recursive types are read-only in VT-RecI (Figure 9 in the appendix). It was not based on subtyping, but rather on the relation **ro**, which makes a read-only version of a type using a syntax-based type splitting.

This definition did not guarantee that all supertypes of a read-only type are also read-only. As we will explain in Section 4.1.3, this would be a critical problem for the SEF guarantee.

We solve this problem by using a different notion of read-only types, based on subtyping with the “read-only bottom” type **N**.

The purpose of N in the original roDOT was to be the read-only version of the type \perp for defining the **ro** relation. Because the bottom type \perp is a subtype of all types, it is inherently mutable. For that reason, the type N was added and made a lower bound of read-only types. That allows us to define read-only types as supertypes of N .

► **Definition 8** (Read-only types). *A type T is read-only, if $\Gamma; \rho \vdash \mathsf{N} <: T$.*

With Definition 8 settled, we discovered a few problems related to read-only types, which would not allow us to state the SEF guarantee in the original roDOT.

Our proof of the SEF guarantee, specifically Lemma 19 in Section 5.4, relies on the idea that if a reference has some read-only type, then any other reference to the same object has that type too. Note that because of subsumption, a variable of a mutable type also has the corresponding read-only types. This essentially means that in any place where a reference is used by virtue of its read-only type, it can be replaced with a read-only version of that reference. With the new Definition 8 of read-only types, this can be stated as:

► **Lemma 9** (Read-only types are shared by all references). *If $\Gamma \sim \rho$ and $\Gamma; \rho \vdash y : T$ and $\Gamma; \rho \vdash \mathsf{N} <: T$, then $\Gamma; \rho \vdash w : T$ for any w such that $\rho(w) = y$.*

This key lemma, however, does not hold in the original roDOT, because of union types.

Union types were not a part of DOT, but were added to roDOT in order to be used to define viewpoint adaptation (union types are already a part of Scala’s type system), along with the subtyping rules ST-Or, ST-Or1, ST-Or2, ST-Dist, which are shown in Figure 10 in the appendix. However, using unions, it is possible to construct a type that is a supertype of both N and a mutability declaration:

► **Example[†] 10** (In the original roDOT, counter-example to Lemma 9). Let $T_{\text{am}} := \{a : T_a\} \vee \mathcal{M}_{\perp}$ be a union of some field declaration with a declaration of mutability, and $T_{\text{bm}} := \{b : T_b\} \wedge \mathcal{M}_{\perp}$ be a type of a writeable reference to some other field b .

The type T_{am} is not mutable, because it is not a subtype of \mathcal{M}_{\perp} . It is read-only, because $\Gamma \vdash \mathsf{N} <: T_{\text{am}}$, by the rules of subtyping of union types and by rule ST-N-Fld (Figure 10 in the appendix).

Let y_1 be a location of type T_{bm} , and w_2 be a reference to y_1 with type $T_b := \{b : T_{\text{bm}}\}$. By subtyping of unions and intersections, $\Gamma \vdash T_{\text{bm}} <: T_{\text{am}}$, so by subsumption, y_1 has type T_{am} . By Lemma 9, w_2 should have also type T_{am} , but in the original roDOT, it does not.

Example[†] 10 is marked with the [†] sign, which we use in this chapter to identify properties of the original roDOT from prior work, in contrast to the modified roDOT in this paper.

We observe that the read-only type T_{am} in this counter-example is a union of disjoint declarations, so it does not allow accessing the field a_{am} , or any other member. Therefore, T_{am} is no more useful for typing programs than \top . In order to make Lemma 9 work, we decided to extend the type system with new subtyping rules to make types like this equivalent to \top . These changes will be described in Section 4.2.

4.1.2 The SEF condition

A method is statically SEF if the types of its receiver and parameters are read-only according to Definition 8 i.e., they are supertypes of N . Thanks to subsumption and subtyping of method types, the type $\{m(z : \mathsf{N}, r : \mathsf{N}) : \top\}$ is a type bound for methods named m and requires that both the argument and receiver have read-only types.

► **Definition 11** (Static SEF condition). *A method is statically SEF if it has a type $\{m(z : \mathsf{N}, r : \mathsf{N}) : \top\}$*

$$\frac{\Gamma; \rho \vdash N <: T}{\Gamma; \rho \vdash T \text{ ro } T} (\text{TS-N}) \quad \frac{}{\Gamma; \rho \vdash \top <: N \vee \{M(r) : T_1..T_2\}} (\text{ST-NM})$$

■ **Figure 4** New rules for roDOT.

In Section 5, we will show that this condition works because a method must access all objects through the argument or the receiver (capturing values is modeled using fields of the receiver), so the method will not be able to get a writeable reference to any existing object.

4.1.3 Subtyping of method types

In order for the SEF guarantee (Theorem 16) to work with Definition 11, it is critical that all subtypes of a SEF method type are also SEF. The reason is at the site of a method call, the observed static type of the method is a supertype of the actual type of the method within its containing object, so this is needed to make the connection from the SEF type at a call site to the SEF type of the actual method.

That is why Definition 8 needs to be based on subtyping, so that all supertypes of read-only types are read-only (method types are contravariant in their parameter types).

Still, the type system required one more change related to a possible dependency between the types of method parameters. In roDOT, the type of the receiver r can be a dependent type referring to the other parameter z of the method. If, however, the receiver type depended on the mutability of z , then while typing the body of the method, it would be possible to derive that z is mutable, even if its type is read-only in the sense of Definition 8. If r has the type $\{A : z.M..\perp\}$, one can use the typing rules ST-SelL and ST-SelU (Figure 10 in the appendix) to derive $z.M <: \perp$. The change to the rules TT-Call and ST-Met in Figure 2 prevents this issue by disallowing using method types where the receiver depends on the mutability of the parameter.

4.2 The updated roDOT calculus

In the previous section, we defined the static SEF condition, but identified several reasons why this definition would not work as intended in roDOT as-is. We fix these issues by changes to the roDOT calculus, which amount to two new and one modified typing rule:

- A new subtyping rule ST-NM (Figure 4) is added, which makes the union of a mutability declaration and the read-only lower bound N a top-like type (the other direction of subtyping was already a part of the type system).
- A new rule TS-N (Figure 4) is added to type splitting, making it so that all types that are read-only by Definition 8 are unaffected by the splitting operation.
- The typing rule TT-Call and subtyping rule ST-Met have a new premise (shown highlighted in Figure 2), which disallows introducing a dependency between the receiver type and the parameter in method subtyping. This fixes the problem described in Section 4.1.3.

The new rule ST-NM fixes the counter-example to Lemma 9, because now we have $\Gamma; \rho \vdash \top <: T_{am}$, derived from $\Gamma; \rho \vdash \top <: N \vee M_\top$ and $\Gamma; \rho \vdash N <: \{a : T_a\}$. By subsumption and $\Gamma; \rho \vdash w_2 : \top$, that also means that $\Gamma; \rho \vdash w_2 : T_{am}$.

Additionally, we can now improve the type splitting relation $\vdash \text{ ro }$, by extending it with a new rule TS-N, shown in Figure 4. With that, the condition in VT-RecI (Figure 9 in the appendix) that recursive types are read-only, $\Gamma; \rho \vdash T \text{ ro } T$, becomes equivalent to Definition 8:

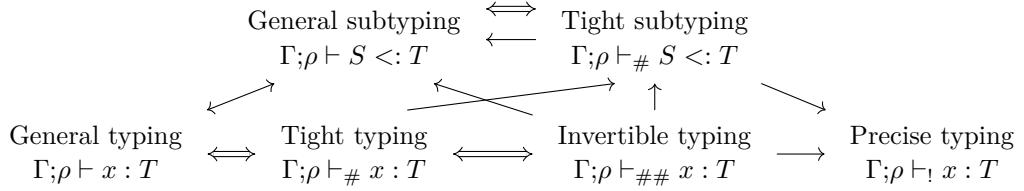


Figure 5 Dependencies (\rightarrow) and equivalences (\leftrightarrow) between definitions of typing in roDOT.

► **Lemma 12** (Read-only types). $\Gamma; \rho \vdash N <: T \Leftrightarrow \Gamma; \rho \vdash T \text{ ro } T$.

4.2.1 Updating the safety proof

The changes described above require updating the type safety proof of the calculus, to show that the changes did not allow invalid programs to be typed. The new subtyping rule ST-NM has a significant effect on the soundness proof, because it makes it possible to derive many additional union types, such as the now top-like type $M \vee N$.

The proof of soundness of roDOT before these changes followed the structure of the proof of DOT [30]. The core part of this proof is to show that if a reference w has some declaration type D (such as a field $\{a : T\}$), then the type associated with w in the typing context Γ is an object type containing D or a more precise declaration of the same member. That means, for $\Gamma; \rho \vdash w : D$, where D is a declaration type, because the types in Γ correspond to the object on the heap ($\Gamma \sim \Sigma$), the actual object referred to by w must contain a corresponding member definition in Σ , and therefore it is safe to access that member.

The proof was based on two alternative definitions of typing for variables – tight typing and invertible typing. Figure 5 shows the relations between the different versions of typing.

Tight typing is used as an intermediate step in equivalence of general and invertible typing. It is very similar to general typing – it has the same rules, except that subtyping rules involving selection types (ST-SelL and ST-SelU in Figure 10 in the appendix) use precise typing, a simpler version of variable typing, which does not have subsumption.

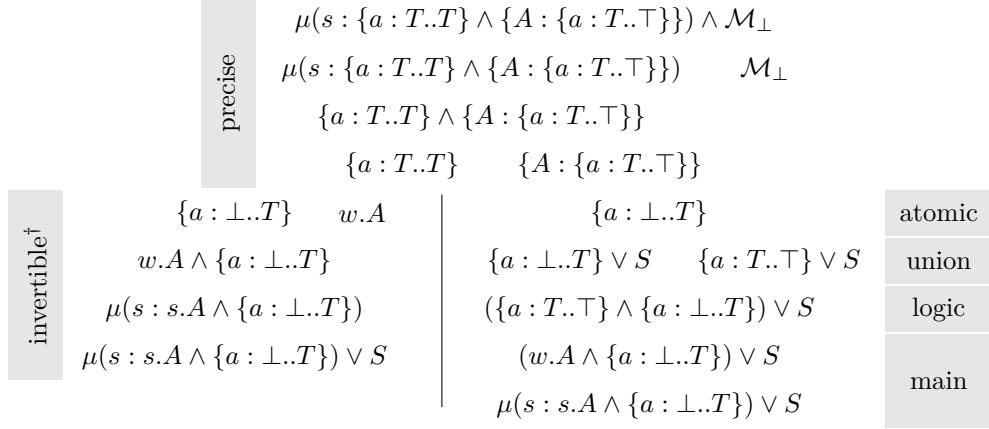
Updating tight typing for our modified rules is straightforward – we apply the same changes as to general typing, and the proof of equivalence between general and tight typing still works. However, we will show that updating invertible typing poses a challenge, as it cannot be easily extended with the additional rules.

4.2.2 Invertible Typing[†]

The main utility of invertible typing was providing a simple path of derivation of a variable's type, starting from the type given to it by the typing context, and ending with a type that was used to access a member at some particular point in the program. This direct path then allowed induction-based proofs of properties of the typing relation.

This task would be especially hindered if the typing rules allowed cycles in the derivation, which would allow the derivation to go through unnecessarily complicated types. For example, with general typing, it is possible to derive $\Gamma \vdash x : T$ from $\Gamma \vdash x : T \wedge T$ and vice versa. Therefore, a derivation of type T can start with $\Gamma \vdash x : T$, go through arbitrarily complicated types such as $(T \wedge T) \wedge T$, and come back to T . This inhibits arguments by induction on the derivation of a general typing.

Invertible typing in DOT prevented this by ensuring that the derivation closely follows the syntactic structure of the target type.



■ **Figure 6** Example derivation of a type by invertible typing (left) and layered typing (right). Assuming that a variable w has the type $\mu(s : \{a : T..T\} \wedge \{A : \{a : T..\top\}\}) \wedge \mathcal{M}_\perp$ in the typing context, w has all the types shown here, ordered from types that are simple to derive at the top, to more complex derivations, which make use of derivations above. S is an arbitrary type.

The original roDOT adopted the invertible typing from DOT [30], where it has two layers, which we present using an example derivation of a type for a variable w in Figure 6.

The first layer, **precise typing**, only derives types by deconstructing the type of the variable given by the typing context. For each reference w , its type in the typing context is an intersection of a mutability declaration with a recursive type containing an intersection of declarations. Precise typing allows opening this recursive type and extracting the declarations from the intersection, but does not support subtyping. The top of Figure 6 shows individual steps of this process.

The second layer, **invertible typing[†]**, combines both variable typing and subtyping into a single layer. In DOT and the original roDOT, it has fewer rules than general typing and subtyping, because it only has rules that construct the target type syntactically “bottom-up”, such as closing recursive types (akin to VT-RecI), or deriving intersection and union types. Thus the derivations of invertible typing are unambiguously guided by the syntax of the target type. The left side of Figure 6 shows individual steps of this process in building up the type $\mu(s : s.A \wedge \{a : \perp..T\}) \vee S$ for w .

As per Figure 5, invertible typing was equivalent to tight typing. That required invertible typing to be closed under tight subtyping (Lemma[†] 13).

► **Lemma[†] 13** (In original roDOT, invertible typing is closed under subtyping).

If $\Gamma; \rho \vdash_{\#} x : T_1$, and $\Gamma; \rho \vdash_{\#} T_1 <: T_2$, where $\Gamma \sim \rho$, then $\Gamma; \rho \vdash_{\#} x : T_2$.

The addition of ST-NM, together with the rules ST-Or and ST-Dist (Figure 2), breaks this. In Lemma[†] 13, the case for ST-Or relies on case analysis of deriving union types (Lemma[†] 14).

► **Lemma[†] 14** (In original roDOT, typing with union types can be inverted).

If $\Gamma; \rho \vdash_{\#} x : T_3 \vee T_4$, then either $\Gamma; \rho \vdash_{\#} x : T_3$ or $\Gamma; \rho \vdash_{\#} x : T_4$.

However, the rule ST-NM adds new ways of deriving union types such as $N \vee M_\perp$, and the distributivity rule ST-Dist actually allows deriving arbitrarily large types of the form $T_N \vee T_M$, where the two parts can consist of arbitrary intersections and unions of various types that contain N and M somewhere within them.

For example, with the variables from Example[†] 10, we have $\Gamma; \rho \vdash w_2 : \{a : T_a\} \vee M_{\perp}$, but $\Gamma; \rho \not\vdash w_2 : \{a : T_a\}$ and $\Gamma; \rho \not\vdash w_2 : M_{\perp}$. Such a type cannot be derived in a syntactically bottom-up manner that invertible typing is based on. Rather than trying to fix invertible typing by adding complicated rules, we replace it by a new auxiliary typing judgment, which derives types in different way.

4.3 Layered Typing

In *layered typing*, we avoid the need for Lemma[†] 14 by organizing the derivation of a type not bottom-up, but by handling different type constructors in separate layers of typing judgments. All union type constructors are derived before intersection types, recursive types and type selections. Additionally, we derive union types on two layers:

First, the *basic layer* derives the newly top-like types possible by the rule ST-NM. Because intersections, recursive types and selections are out of the picture at this layer, these types have a simple form of possibly nested union types, where one of the sides contains N and the other M , where M is a declaration $\{M : \perp..T\}$ for some bound T . We will write that as $\vdash N \triangleleft T_N$ and $\vdash M \triangleleft T_M$. Second, the *union layer* derives types possible by the rules ST-Or1 and ST-Or2, allowing nesting a known type of w in a union with any other type. This way, the layers retain the information about how a union type has been derived and those cases can be handled separately when inverting the derivation.

Intersection types can be handled in analogy to how any logical formula can be derived by starting from conjunctive normal form (CNF) and pushing conjunctions down. Any type constructed from a mixture of union and intersection types can be derived by starting from an intersection of union types and pushing the intersections down.

The *logic layer* sitting above the union layer can derive any mixture of unions and intersections using the LTL-And rule shown in Figure 7. It takes derivations of two types that may have some parts in common but differ in one place. The common part C^{\vee} is a syntactical context which combines the argument into a union with other types. For example, we can write the two types $\{a_1 : T_1\} \vee \{a_2 : T_2\}$ and $\{a_1 : T_1\} \vee M_{\perp}$ as $C^{\vee}[\{a_2 : T_2\}]$ and $C^{\vee}[M_{\perp}]$. If we view these two types as an intersection, then the rule pushes the intersection down to the place where the two types differ. In the example derivation on the right of Figure 6, we derived two union types on the union layer. (The type $\{a : T..T\}$ was derived on the previous layer and S is an arbitrary type.) On the logic layer, we combined them into one type, pushing the intersection down to the left.

The rest of the type constructors are handled either below the basic layer or above the logic layer. Subtyping between declarations is handled in an *atomic layer* positioned before the basic layer. This layer only deals with types that are single declarations.

Recursive types of the form $\mu(s : T)$ and selection types can “wrap around” or replace any part of the derived type (in general typing by VT-RecI and ST-SelL, Figures 9 and 10 in the appendix), which may both appear under unions and intersections, and also contain them within. Therefore, they are handled above the logic layer in a final, *main layer*. In the rule LTM-Sel in Figure 7, the syntactic context $C^{\wedge\vee}$ can consist of a mixture of unions and intersections. The rules have premises that correspond to conditions in the relevant rules of tight typing. For example, in Figure 6, the left side of the union is wrapped under a recursive type in the last step.

The layers are summarized in Table 1, showing the relevant type constructors and the connection to rules of general typing. Selected rules are shown in Figure 7; full definitions are in Figures 15–19 in the appendix.

■ **Table 1** The layers of layered typing.

Typing layer	Relevant type constructors	Relevant rules
Atomic layer	$\{a : T..U\}, \{A : T..U\}, \{m(S, T) : U\}$	ST-Met, ST-Fld, ST-Typ
Basic layer	$N \vee M$	ST-NM
Union layer	$T, T \vee U$	ST-Or1, ST-Or2, ST-Top
Logic layer	$T \wedge U$	ST-Dist, ST-And, VT-AndI
Main layer	$\mu(s : T), x.A$	VT-RecI, ST-SelL, ST-N-Rec

$$\frac{\Gamma; \rho \vdash_1 x : C^\vee[T_1] \quad \Gamma; \rho \vdash_1 x : C^\vee[T_2]}{\Gamma; \rho \vdash_1 x : C^\vee[T_1 \wedge T_2]} \text{(LTL-And)} \quad \frac{\Gamma; \rho \vdash_m x : C^{\wedge\vee}[[v_3/r]T_1] \quad \Gamma \vdash_! v_2 : \{B(r) : T_1..T_2\}}{\Gamma; \rho \vdash_m x : C^{\wedge\vee}[v_2.B(v_3)]} \text{(LTM-Sel)}$$

■ **Figure 7** Selected rules of layered typing.

- Typing on the **atomic layer** ($\Gamma; \rho \vdash_a x : T$) only gives variables single declaration types – the declarations derived by precise typing, and their supertypes (subtyping rules between declarations are handled here).
- The **basic layer** ($\Gamma; \rho \vdash_b x : T$) additionally gives all variables top-like types of the form $T_N \vee T_M$ and $T_M \vee T_N$, where $\vdash N \triangleleft T_N$ and $\vdash M \triangleleft T_M$.
- The **union layer** ($\Gamma; \rho \vdash_u x : T$) handles \top and unions of known and arbitrary types.
- The **logic layer** ($\Gamma; \rho \vdash_l x : T$) handles intersections and distributivity. The rule LTL-And takes two types, preserves their common part, and combines the differing parts using an intersection type – pushing the intersection down from the top to its target place.
- The **main layer** ($\Gamma; \rho \vdash_m x : T$) closes recursive types and handles type selections.

For layered typing, we also proved the following properties:

- If a location has a declaration type by layered typing, then it also has a declaration type by precise typing, with the same or tighter bounds. This property has three variants, for field, type and method declarations.
- Layered typing is equivalent to general typing. As in the original proof, we use tight typing as a step between general and layered typing, and separately prove both directions of equivalence between tight and layered typing (Lemma 31 and Lemma 37 in the appendix).
- We also use layered typing to prove Lemma 9 – if a location has some read-only type in layered typing, then all references to that location have that type too.

With these properties, the safety proof from roDOT, with invertible typing replaced by the new layered typing definition, works as a safety proof of the updated calculus. Formal statements of these and other selected properties are given in Section A.3 in the appendix.

5 The SEF Guarantee

In Section 3.3, we informally stated the SEF guarantee, which provides the connection between a static typing condition (Definition 11) and run-time behavior of the method.

In this section, we present the formal definitions of the run-time SEF condition (Definition 15 in Section 5.1) and the SEF guarantee (Section 5.2). We then outline the proof of the guarantee (Section 5.3) and discuss some details of the proof (Section 5.4).

5.1 The run-time SEF condition

We informally stated the run-time SEF condition in Section 3.1, where we mentioned that several possible versions of such a condition could be defined. In our approach, we allow a pure method to create new objects and to modify just these new objects, which are under full control of the method.

The main SEF condition is that the method must not modify any **existing objects** that are already on the heap when the method starts executing. We can state such a condition in three variants, depending on the way in which it is checked that an object was not modified. Here we will use the variant that guarantees that existing objects on the heap do not change. In such a case, we say that a given execution of a method, starting from a method call start and reaching method call end in k steps, has the Sef-I property (Definition 15). The other possible variants are stated as Definition 42 and Definition 43 in the appendix.

► **Definition 15 (Sef-I).** A method execution $\langle w_1.m\ w_2; \sigma; \rho_1; \Sigma_1 \rangle \xrightarrow{k} \langle vw_3; \sigma; \rho_2; \Sigma_2 \rangle$ is Sef-I when for every $j \leq k$ and $\langle w_1.m\ w_2; \sigma; \rho_1; \Sigma_1 \rangle \xrightarrow{j} \langle t_3; \sigma_3; \rho_3; \Sigma_3 \rangle$, Σ_1 is a prefix of Σ_3 .

5.1.1 Method call limits

Because we are defining a condition on what can happen while a method is executing, we need to understand what it means in roDOT that a method starts and ends its execution.

In roDOT, a method is called by a term $w_1.m\ w_2$. A *method call start* is a configuration of the form $\langle w_1.m\ w_2; \sigma; \rho_1; \Sigma_1 \rangle$, where w_1 is the receiver, m is the called method, w_2 is the argument, σ is the *continuation stack*, and Σ_1 is the *existing heap* (the environment ρ_1 does not have a special significance here).

The execution proceeds by replacing the call term $w_1.m\ w_2$ with the body of the method. Then, the body is executed. Unless there is an infinite loop, the body of the method will eventually evaluate to a single value. The machine will reach a configuration $\langle vw_3; \sigma; \rho_2; \Sigma_2 \rangle$, where w_3 is the result of the call and σ is the same stack as at the method call start.

The **first** such configuration after a method call start is the corresponding *method call end*. Another such configuration could possibly occur later in a completely unrelated way, but only the first such configuration is the method call end.

When a method call end is reached, the execution will either terminate, or proceed by popping a frame from the stack.

5.2 The SEF guarantee

The SEF guarantee, informally stated in Section 3.3, says that a SEF method does not modify existing objects in the heap during its execution. Theorem 16 is based on Definition 15, and speaks about the state of the heap at every point during the call. It is not the strongest possible purity guarantee, because this allows writing the value that already is in the field. On the other hand, it does not allow the value of fields to be changed and then changed back.

► **Theorem 16.** Let the configuration $c_1 := \langle w_1.m\ w_2; \sigma_1; \rho_1; \Sigma_1 \rangle$ be well-typed in a context Γ . Further assume that $\Gamma \vdash w_1 : \{m(z : \mathbb{N})(r : \mathbb{N}) : \top\}$. Then for any k steps of execution:

1. Either the method call has finished executing. There is $j < k$ for which $c_1 \xrightarrow{j} \langle vw_3; \sigma_1; -; - \rangle$.
2. Or, the method call has not finished executing and in this period existing objects in the heap are unchanged. For each $c_1 \xrightarrow{k} c_2$, all heap locations in c_1 also exist in c_2 and moreover they are unchanged in c_2 .

5.3 Overview of the proof

The SEF guarantee talks about objects not being modified during the execution of methods, based on the mutability of method parameters. We base our proof of the SEF guarantee on the immutability guarantee (IG, Theorem 2), which states that individual objects can only be modified through mutable references. This guarantee was proven for roDOT [12] and is included in our mechanization in Coq.

However, the immutability guarantee cannot be applied at the start of the method, because there may be many mutable references to objects on the heap. Also, IG guarantees immutability until the end of execution of the whole program, but the SEF guarantee only until the end of the method.

These differences can be bridged by taking the machine configuration at the method start, and constructing a different configuration that will execute the same way until the end of the method, but removing the parts that prevent the IG from applying.

First, note that the stack is not relevant to how the method executes and stays the same from the method start until its end. We therefore remove this stack entirely, and get an execution isolated from the rest of the program. This execution proceeds through the same steps, but stops at the method end. By removing the stack, we rid the configuration of any references to objects that might be used after the method call returns. If we apply the IG to this configuration, it will guarantee that objects are not modified until the end of the method, exactly what is needed for the SEF guarantee.

Removing the stack is not enough for the IG to apply though, because a SEF method can be called with arguments that are mutable references. We do not want to prevent that from happening, because even when a method is SEF, it can be useful to pass mutable references to the method and have it return one of these references with its mutability intact.

What is special about a SEF method is that (because of its declared parameter types) it cannot use the mutability during its execution. Therefore, when called with mutable arguments, it should execute in exactly the same way as if called with read-only arguments.

So the second modification to the configuration after removing the stack is to change the mutability of the arguments to read-only. That way, the alternative configuration contains no writeable references, and IG guarantees that no objects that were on the heap at the start will be modified. Still, this alternative configuration executes the same steps as the original, meaning the original method execution also does not modify any existing object on the heap.

5.4 Proof of the SEF Guarantee

The strategy of the proof of Theorem 16 is to focus on the second case of the SEF guarantee by using the immutability guarantee to show the theorem for a configuration c_2 obtained by temporarily truncating the stack of c_1 from Theorem 16.

► **Lemma 17** (SEF guarantee without stack). *For c_1 satisfying the conditions of Theorem 16, let $c'_1 := \langle w_1.m\ w_2; \cdot; \rho_1; \Sigma_1 \rangle$. If $c'_1 \mapsto^n c'_2$ for some n and c'_2 , then the heap of c'_2 contains all objects of c'_1 without modification.*

It is easy to prove the full SEF theorem with this result for c'_1 . The premise of the immutability guarantee is that c'_1 is well-typed in some context Γ_2 and there are no mutably reachable objects in c'_1 with respect to the typing of Γ_2 . Clearly c'_1 is well typed in the original context Γ . But for the part about mutably reachable objects, we cannot just take Γ as Γ_2 because for this, Γ_2 must assign read-only types to w_i . Even though we have $(r : \mathbb{N})$ in the typing $\Gamma \vdash w_1 : \{m(z : \mathbb{N})(r : \mathbb{N}) : \top\}$, this does not necessarily mean $\Gamma(w_1)$ is a read-only type. For example, w_1 might be mutable in Γ but m might not make use of its mutability. Therefore, instead of using Γ as Γ_2 , we construct Γ_2 and show that c'_1 is well-typed in Γ_2 like so:

1. *Reference elimination.* Remove bindings for w_i from Γ and replace all occurrences of w_i with the corresponding location y_i in both Γ and c'_1 .
2. *Read-only weakening.* Add back bindings for w_i , where the new type bound to w_i is the type bound to y_i except with the mutable part set to read-only.

We do this instead of changing the types of w_i , because we only know that w_i are used in a read-only way in the focus, while on the heap, w_i might be used as a part of dependent types referring to their mutability. Changing their types would break the typing of the heap.

The second step is essential, because only references can be read-only, while locations always have mutable types. The references w_i are added in the same order as in the original context, to ensure that types in the typing context only refer to preceding variables in case the types are dependent. The two steps above correspond to the following two lemmas.

► **Lemma 18** (Reference elimination). *Let c_1 and Γ satisfy the conditions of Theorem 16, and $\rho_1[w_i] = y_i$. Define Γ' as the context obtained from Γ by first removing bindings for w_i and then replacing w_i with y_i . Define ρ' as the environment obtained from ρ_1 by removing bindings for w_i . Then the term $y_1.m\ y_2$ is well-typed under $\Gamma'; \rho'$ and we have heap correspondence $\Gamma'; \rho' \sim [y_i/w_i]_i \Sigma_1$.*

Proof. Because w_i is a reference to y_i , types assigned to y_i and w_i by Γ differ only by mutability, and y_i has a mutable type. So $\Gamma(y_i)$ is a subtype of $\Gamma(w_i)$, and the result follows by substitutivity. ◀

► **Lemma 19** (Read-only weakening). *Let c_1 and Γ satisfy the conditions of Theorem 16, and y_1, y_2 be such that $\rho_1[w_i] = y_i$. Then there is a context Γ_2 binding w_i to read-only types such that the configuration $c''_1 := \langle w_1.m\ w_2; \cdot; \rho_1; [y_i/w_i]_i \Sigma_1 \rangle$, is well-typed in Γ_2 (formally $\Gamma_2 \vdash c''_1 : \top$).*

By Lemma 19 along with the immutability guarantee, we have SEF established for $c''_1 := \langle w_1.m\ w_2; \cdot; \rho_1; [y_i/w_i]_i \Sigma_1 \rangle$. However, we need SEF in particular for the configuration $c'_1 := \langle w_1.m\ w_2; \cdot; \rho_1; \Sigma_1 \rangle$ in Lemma 17, where there is no substitution $[y_i/w_i]_i$ in the heap. Nevertheless, this substitution can be ignored in the sense that execution can only change fields of objects, but in roDOT, fields always store locations while w_i are references. (When a reference is assigned to a field, the corresponding location is stored, in order to make the field type determine the mutability of the stored value.) Because c'_1 and c''_1 are the same everywhere except for $[y_i/w_i]_i$ on the heap, the SEF property of c''_1 can be carried over to c'_1 . The first part of carrying the SEF property over to c'_1 is to relate each k -th step of execution starting from c''_1 and the k -th step of execution starting from c'_1 . We need to show that the two executions are almost the same, except some specific variables that appear in the machine configuration can differ. In particular, the references w_i can be replaced by the locations y_i . Additionally, the locations and references of newly created objects can differ, because variable names are not assigned deterministically.

For that reason, we define the *similarity* relation, which formalizes structural equivalence of syntactic elements such as terms, objects or whole configurations that differ only in names of variables. It relates two such elements using a *renaming* relation over variables. A formal definition of similarity and its basic properties is given in Section A.5 in the appendix.

Similarity has two important properties with respect to program execution: (1) it is preserved by reduction, and (2) if a machine configuration can reduce, then all similar configurations can reduce too (and the results are similar). That means that if we start with two similar configurations, where the execution of one reaches an answer state, then the execution of the other will reach a similar answer state. With this definition of similarity, Lemma 20 formalises the idea that c''_1 is similar to c'_1 up to renaming w_i to y_i :

► **Lemma 20** (Similarity for eliminated references). *Let c'_1 satisfy the conditions of Lemma 17 and c''_1 the conditions of Lemma 19. Then $c'_1 \xrightarrow{(w_1,y_1),(w_2,y_2)} c''_1$.*

The final part of carrying over the SEF property to c'_1 is to recognize that reduction only changes values of fields (while the structure of the object, methods and type members are immutable).

► **Definition 21** (Objects identical except fields). *For objects o_1 and o_2 , we write $o_1 \xrightarrow{fld} o_2$ to mean they are identical except for possibly the values of fields.*

► **Lemma 22** (Reduction only changes fields). *If $\langle -; -; -; \Sigma \rangle \mapsto^n \langle -; -; -; \Sigma' \rangle$ and y is a location in Σ , then $\Sigma(y) \xrightarrow{fld} \Sigma'(y)$.*

And with this, we can finish the proof of Theorem 16.

Proof. By classical reasoning, assume the condition 1 is false so that the goal is to prove the condition 2. That is, assume that there is no $j < k$ such that the top-most frame of c_1 is popped after execution by j steps: $c_1 \mapsto^j \langle vw_3; -; -; \Sigma_2 \rangle$. Then, the sequence of reductions $c_1 \mapsto \dots \mapsto c_2$ corresponds to a sequence of reductions $c'_1 \mapsto \dots \mapsto c'_2$ because even though c'_1 has no awaiting frames, there are no frame pops in this execution sequence by the current assumption. By Lemma 17, the condition 2 follows. ◀

6 Transformations

In Section 3.4, we informally stated the transformation guarantee, which connects the static SEF condition with a practical application – that calls to SEF methods can be safely swapped.

Defining the call-swapping and the guarantee in a formal way requires dealing with several technicalities particular to the roDOT calculus (or DOT calculi in general). In order to separate the common problems from the specific case of swapping calls, we build a general framework in which various transformations of roDOT programs can be defined and proven safe. We instantiate it here only with the call swapping transformation, but it could represent the general part of a proof of safety for other transformations, such as reordering field reads or removing dead code.

In Section 6.1, we present the framework for defining and reasoning about safe program transformations in roDOT and similar calculi, including a general Theorem 25 about safety of transformations. In Section 6.2, we define the transformation that swaps two calls to methods that are statically determined to be side-effect free (Definition 26) and state the transformation guarantee.

6.1 Transformation framework

The framework defines a general form for roDOT program transformations, defines the precise meaning of a *safe transformation* that “does not affect the behavior of the program”, and provides a way of proving this property, while helping to deal with common technical issues of DOT formalizations.

Our general approach is to define a transformation that applies to an initial program, and prove it safe by showing that if the original and transformed program are executed side-by-side, they will either eventually reach the “same” answer, or both not terminate.

In the initial program, a transformation, such as swapping calls, can be located anywhere, including inside a body of a method of an object literal, such as $\text{let } x = \nu(r : R)\{m(r, z) = t_m\} \text{ in } t_2$. We must consider that in roDOT, terms are in A-normal form, and that a term can

have multiple different types. Also code (terms) and values (objects) are mixed with each other during execution of a program – the program contains object literals, and methods on the heap contain program code.

During execution, objects are created on the heap, including copies of the code of their methods, which can be affected by the transformation. It would be too restrictive to require that a transformed program produce the exact same output value as the original program since the output value may be an object that may contain the transformed code. To facilitate this, we define each transformation using a local relation which relates two terms that differ only locally, and the framework provides lifting operators, which allow this transformation to occur anywhere in a program or in a machine configuration.

The general safety Theorem 25 is based on executing the two programs and observing that the intermediate states are also related by the transformation (lifted to whole configurations and allowed to occur at multiple places), except the moments when the directly affected part of the program is executing. When the two answers are reached, they will be similar except that bodies of methods on the heap may differ as the transformation permits.

A more detailed description of the framework design and its definitions are given in Section A.7 in the appendix. The following text describes the most important parts.

6.1.1 Transformations of roDOT programs in general

A transformation of a program is defined as a binary relation on terms – the original program and the transformed one. For example, the call-swapping transformation is defined as a relation that relates a program containing two method calls with a program that only differs in the order of those calls.

Because the safety of the transformation depends on typing information, we define the transformation as a binary relation between triples: the term, its type and a typing context. This generalizes to other syntactic elements – stacks, heaps and machine configurations, though for each kind of element, the exact meaning of “typing context” and “type” may differ. For terms, the typing context is actually paired with a runtime environment $\Gamma; \rho$.

► **Definition 23** (Transformation). *A transformation τ is a relation between triples consisting of typing contexts $\Gamma_{1,2}$, types $T_{1,2}$ and typeable elements $X_{1,2}$. We write $\langle \Gamma_1 \vdash X_1 : T_1 \rangle \rightarrow_\tau \langle \Gamma_2 \vdash X_2 : T_2 \rangle$ and say that X_1 is transformed into X_2 .*

Like any binary relation, transformations can be symmetric, reflexive or transitive, and we can construct transformations using iteration, composition, union and inversion.

Additionally, a transformation is *type-safe*, if the syntactic elements on both sides are correctly typed under the respective contexts. Another useful property of a transformation is being *type-identical*, where both the types and typing contexts are the same on both sides.

To facilitate the possibility of the transformation being located anywhere in a term, it is useful to define the transformation in two steps: (1) A *local transformation*, which only allows swapping calls at the root of the term. (2) A lifting operator $\text{lift } \tau$ which takes a local transformation τ and allows it to be located at one place anywhere in a term.

Such a local transformation of a term can be further lifted by $\text{cfg } \tau$ to a whole run-time configuration, where τ applies at exactly one place in the focus of execution, in the stack or on the heap. To allow multiple occurrences, we can apply the iteration operator to the lifted transformation. Having a definition that only allows one occurrence is useful in the proof of Theorem 25 in Section A.7.4 in the appendix, where we want to look at each occurrence individually. More details about lifting are in Section A.7.3; the definitions of the lifting operators are shown in Figure 23 and Figure 24.

6.1.2 Safe transformations

The definition of a safe transformation must allow the answers of the two programs to contain different variable names and allow for the fact that the transformation can occur in the heap of the program answer, possibly at multiple places. Therefore a local transformation is safe if execution of the transformed program reaches answers related by an iteration of this transformation. To deal with non-deterministic location names, the transformation is in union with a *similarity transformation* \approx , which allows one-to-one variable renaming.

► **Definition 24** (Safe transformation). *A transformation τ is safe if $\langle \Gamma_1 \vdash c_1 : T \rangle \rightarrow_\tau \langle \Gamma_2 \vdash c_2 : T \rangle$ and $c_1 \rightarrow^k c_3$, where c_3 is an answer typed as $\Gamma_3 \vdash c_3 : T$, implies that there exist c_4 , Γ_4 and j such that $c_2 \rightarrow^j c_4$, $\Gamma_4 \vdash c_4 : T$ and $\langle \Gamma_3 \vdash c_3 : T \rangle \rightarrow_{(\tau \cup \approx)^*} \langle \Gamma_4 \vdash c_4 : T \rangle$.*

Thanks to being able to define a transformation by applying a general lifting to a local transformation, the safety proof of such a transformation can be also divided into a theorem that will apply to any local transformation with certain local properties, and then proving those local properties for the particular local transformation.

This approach makes it possible to state the call-swapping guarantee presented here (Theorem 27), or analogous guarantees for other local transformations.

Theorem 25 states that if a local term transformation does not change typing of the term, is compatible with properties such as weakening, narrowing and substitution, does not change whether the term is an answer or not, and if execution of just the transformed term will eventually reach similar configurations, then transforming a program by this transformation anywhere will not change its result. Full definitions of the premises are in Section A.7 in the appendix as Definitions 49, 50, 52, 56 and 54.

► **Theorem 25** (General safety for local transformations). *If τ is a transformation that is type-identical, type-safe, compatible with weakening, narrowing and substitution, preserves answers, and eventually reduces to similarity, then $(cfg \tau \cup \approx)^*$ is safe.*

6.2 The call-swapping transformation

The specific transformation guarantee that we want to achieve should state that swapping two calls will not change the outcome of the program, in the sense of Definition 24.

Call-swapping is defined as a local transformation that transforms one program containing two successive calls into another program in which the calls are swapped. Due to the A-normal form of terms, two successive calls in the program have the form $\text{let } x_{c1} = x_{o1}.m_1x_{a1} \text{ in let } x_{c2} = x_{o2}.m_2x_{a2} \text{ in } t$. In the transformation, the two calls $x_{o1}.m_1x_{a1}$ and $x_{o2}.m_2x_{a2}$ appear in the opposite order, but the continuation t is the same.

The transformation is only safe if both the methods are SEF, so it has several typing premises, analogous to the ones of Theorem 16 in Section 5.2.

► **Definition 26** (Local call swapping). *The local call-swapping transformation csw is a transformation of terms that relates $\langle \Gamma \vdash \text{let } x_{c1} = x_{o1}.m_1x_{a1} \text{ in let } x_{c2} = x_{o2}.m_2x_{a2} \text{ in } t : T \rangle \rightarrow_{csw} \langle \Gamma \vdash \text{let } x_{c2} = x_{o2}.m_2x_{a2} \text{ in let } x_{c1} = x_{o1}.m_1x_{a1} \text{ in } t : T \rangle$ when*

- $x_{c1,2}$ are distinct from $x_{a1,2}$ and $x_{o1,2}$,
- $\Gamma \vdash x_{o1}.m_1x_{a1} : T_{c1}$, $\Gamma \vdash x_{o2}.m_2x_{a2} : T_{c2}$, and $\Gamma, x_{c1} : T_{c1}, x_{c2} : T_{c2} \vdash t : T$,
- $\Gamma \vdash x_{o1} : \{m_1(r_1 : \mathbb{N}, z_1 : T_{a1}) : \top\}$, and $\Gamma \vdash x_{o2} : \{m_2(r_2 : \mathbb{N}, z_2 : T_{a2}) : \top\}$,
- $\Gamma \vdash x_{a1} : T_{a1}$, and $\Gamma \vdash x_{a2} : T_{a2}$,
- $\Gamma \vdash \mathbb{N} <: T_{a1}$, and $\Gamma \vdash \mathbb{N} <: T_{a2}$.

As the final form of the transformation guarantee, we apply Definition 24 to Definition 26, and specialize the theorem to initial programs. The proof is given in Section A.8 in the appendix.

► **Theorem 27** (Transformation guarantee). *If $\langle \vdash t_1 : T \rangle \rightarrow_{lift\ csw} \langle \vdash t_2 : T \rangle$ and $\langle t_1 ; \cdot ; \cdot ; \cdot \rangle \rightarrow^k c_3$, where c_3 is an answer typed as $\Gamma_3 \vdash c_3 : T$, then there exists c_4 , Γ_4 and j such that $\langle t_2 ; \cdot ; \cdot ; \cdot \rangle \rightarrow^j c_4$, c_4 is an answer typed as $\Gamma_4 \vdash c_4 : T$ and $\langle \Gamma_3 \vdash c_3 : T \rangle \rightarrow_{(cfg\ csw \cup \approx)^*} \langle \Gamma_4 \vdash c_4 : T \rangle$.*

7 Related work

Since the topic of this work includes both the DOT calculus and method purity, here we discuss previous work related to these concepts. Prior to this work, many variants of the DOT calculus were published, some including mechanized proofs. Also the issue of purity in object-oriented languages is of great research interest, and it is approached from different angles of automation and precision. We give details about the existing work in the following subsections. As far as we know, our work is the first one to consider the issue of purity within a DOT calculus.

7.1 Mechanizations of DOT calculi

The first appearance of a DOT calculus [3] did not include a proof of soundness, but was followed by several versions with proofs in Coq [33, 30] and Iris [17]. In particular, WadlerFest DOT [2], thanks to its simplicity and its proof of soundness based on invertible typing [30], was used as a baseline for numerous extensions [32, 22, 31, 23], including roDOT. While objects are immutable in WadlerFest DOT, it was extended to support mutation using mutable slots in Mutable WadlerFest DOT [32], and more directly by allowing changing values of fields in kDOT [22]. A simplified version [21] with mutable fields, but without the specific kDOT feature of constructors, was used as a base for the mechanization of roDOT.

The differences between the mechanization of roDOT and those of previous DOT calculi mainly stem from the differences in how roDOT handles variables – namely, typing of variables and terms being separated from each other, using different definitions of typing contexts to support variable hiding, using the runtime environment to map references to locations, and using typing information in its definition of operational semantics.

The mechanization of roDOT includes a feature to ease further extensions to the calculus. The definitions and theorems are parameterized by a “typing mode”, which allows selecting type system features that are supported. Using this feature, our proofs work for roDOT both with and without the changes described in this paper.

7.2 Purity in other languages

Purity in programming is such an important concept that in many languages, functions are pure by default. This approach is typically associated with functional programming, but an object-oriented system can also be pure [1] when the objects are immutable. That is also the case in the basic DOT calculus.

In pure functional languages, effects must typically be explicitly declared in the program using monadic types. This style of programming has been shown to be as powerful as other styles and is used in practical programming languages such as Haskell.

Regarding purity in object oriented languages with mutable fields, many publications [37, 34, 38, 40, 6, 16, 29] focus on Java and languages with similar type systems, such as C#. For Scala, a type system for purity was developed, but not based on the DOT calculus [35].

When approached from a practical standpoint, the definition of purity in these languages has to include considerations other than modification of object fields, such as accessing global variables or synchronization. This leads to different definitions of purity. The term “pure” is sometimes used to mean the same as “side-effect-free”, without requiring determinism.

Observational purity [25, 5] is a weaker property that allows side effects as long as they are not observable from certain parts of the code. This definition is based on classes and access control, features which are not modeled in DOT calculi.

Purity is of great use to program verification and specification frameworks, where it enables inserting run-time checks without changing behavior, and allows more precise analysis. Code Contracts [15], JML [20] and Checker Framework [14] allow annotating a method as pure. Code Contracts do not check that this annotation is correctly applied, and JML and Checker Framework use simpler checks, where pure methods are not allowed to call impure methods. Checker Framework uses the fact that side-effect free methods do not invalidate flow-sensitive types of local variables.

To avoid imposing an annotation burden on the programmer, purity can be inferred by automatic program analysis [26, 34], and side-effect analysis can be used for program optimization [11].

ReIm [19] provides both a type system for reference mutability and a way to automatically infer mutability types. It can therefore automatically find pure methods, which have all parameters read-only. We adopted this way of recognizing pure methods by parameter types for roDOT in this work. While in ReIm, mutability is attached to parameter types as a qualifier in the style of the Checker Framework, roDOT uses the special member type M to include the mutability in the parameter type using intersection types. In ReIm, mutability qualifiers are subjected to qualifier polymorphism and viewpoint adaptation. roDOT can express the equivalent of polymorphic qualifiers using dependent types and implements viewpoint adaptation using union and intersection types [12].

7.3 Capability and Effect Systems

There are other ways to express the permitted side-effects of functions using types, which have been developed in recent work on formal type systems.

The principle of **capabilities** [28, 27] is to require every operation that can have a side effect to take an extra value, called a capability, as a parameter. Then, if some function or method does not have the capability value corresponding to a particular effect, we can conclude that it does not perform that effect. Capabilities are well suited for coarse-grained effects, such as performing input/output in general or accessing some specific file, where a single capability value can guard a set of related operations. To apply such an approach to reasoning about a fine-grained effect such as writing to a field of a specific object, we would need large numbers of such capability values, one new capability value for each existing object. For each reference passed to a parameter or stored in a field, a corresponding capability would need to be passed or stored, thus multiplying the number of parameters and fields.

Wyvern's effect system [24] expresses possible effects by type members of objects. That is syntactically similar to how roDOT represents mutability, but the meaning of the type members is different. In roDOT, the type member of an object reference defines the bounds on the mutability of the reference, the knowledge about whether a reference may be used for mutation, in the type of that reference. In contrast, in Wyvern, the effect member represents

a permission to perform an effect, such as `file.Write`, where the effect can be independent of the object that contains the effect member. Thus, Wyvern effect members are more similar to the capability-based approach.

Another successful direction is to use types to express sets of possible variables captured or aliased by values in the program. **Capture Types** [7] follow from a capability based approach, and enable reasoning about where capability values may be stored in the heap or captured in closures, in order to more precisely reason about where effects may occur. **Reachability Types** [4] annotate the type of an expression with a set of variables, which are values that are possibly reachable from the result of that expression. This can be used in conjunction with effect qualifiers as in Graph IR [8], where a function type declares a set of variables that can be read or written, describing the possible effects in a fine-grained way. The types can also be extended to support qualifier polymorphism [39]. This work is defined in the context of a higher order functional formalism, whereas roDOT is an object-oriented calculus. Also, both Wyvern and Reachability Types express effects using new constructs added to the type system, while roDOT aims to encode mutability using the existing DOT constructs of dependent types, unions and intersections.

8 Conclusion

To conclude, our paper confirms that the reference mutability system provided by roDOT can be mechanically proven sound, and with a few changes can be used to guarantee side-effect freedom of methods, and to justify safe transformations of programs.

References

- 1 Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996. doi:[10.1007/978-1-4419-8598-9](https://doi.org/10.1007/978-1-4419-8598-9).
- 2 Nada Amin, Samuel Grüter, 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](https://doi.org/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*, OOPSLA ’14, pages 233–249. ACM, 2014. doi:[10.1145/2660193.2660216](https://doi.org/10.1145/2660193.2660216).
- 4 Yuyan Bao, Guannan Wei, Oliver Bračevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. Reachability types: Tracking aliasing and separation in higher-order functional programs. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021. doi:[10.1145/3485516](https://doi.org/10.1145/3485516).
- 5 Mike Barnett, David A Naumann, Wolfram Schulte, and Qi Sun. 99.44% pure: Useful abstractions in specifications. In *ECOOP workshop on formal techniques for Java-like programs (FTfJP)*, 2004.
- 6 William C. Benton and Charles N. Fischer. Mostly-functional behavior in Java programs. In Neil D. Jones and Markus Müller-Olm, editors, *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings*, volume 5403 of *Lecture Notes in Computer Science*, pages 29–43. Springer, 2009. doi:[10.1007/978-3-540-93900-9_7](https://doi.org/10.1007/978-3-540-93900-9_7).
- 7 Aleksander Boruch-Gruszecki, Martin Odersky, Edward Lee, Ondřej Lhoták, and Jonathan Brachthäuser. Capturing types. *ACM Trans. Program. Lang. Syst.*, 45(4), November 2023. doi:[10.1145/3618003](https://doi.org/10.1145/3618003).

- 8 Oliver Bračevac, Guannan Wei, Songlin Jia, Supun Abeysinghe, Yuxuan Jiang, Yuyan Bao, and Tiark Rompf. Graph IRs for impure higher-order languages: Making aggressive optimizations affordable with precise effect dependencies. *Proc. ACM Program. Lang.*, 7(OOPSLA2), October 2023. doi:10.1145/3622813.
- 9 The Checker Framework Manual: Custom pluggable types for Java. URL: <https://checkerframework.org/manual/#initialization-checker>, 2022.
- 10 The Checker Framework Manual: Custom pluggable types for Java. URL: <https://checkerframework.org/manual/#purity-checker>, 2022.
- 11 Lars Ræder Clausen. A Java bytecode optimizer using side-effect analysis. *Concurrency: Practice and Experience*, 9(11):1031–1045, 1997. doi:10.1002/(SICI)1096-9128(199711)9:11<1031::AID-CPE354>3.0.CO;2-0.
- 12 Vlastimil Dort and Ondřej Lhoták. Reference mutability for DOT. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*, volume 166 of *LIPICS*, pages 18:1–18:28. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ECOOP.2020.18.
- 13 Vlastimil Dort, Yufeng Li, Ondřej Lhoták, and Pavel Parízek. Pure methods for roDOT (an extended version). Technical Report D3S-TR-2024-01, Dep. of Distributed and Dependable Systems, Charles University, 2024. URL: https://d3s.mff.cuni.cz/files/publications/dort_pure_report_2024.pdf.
- 14 Michael D. Ernst. Annotation type Pure. <https://checkerframework.org/api/org/checkerframework/dataflow/qual/Pure.html>, 2022.
- 15 Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded contract languages. In Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew J. Palakal, and Chih-Cheng Hung, editors, *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26, 2010*, pages 2103–2110. ACM, 2010. doi:10.1145/1774088.1774531.
- 16 Matthew Finifter, Adrian Mettler, Naveen Sastry, and David A. Wagner. Verifiable functional purity in Java. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 161–174. ACM, 2008. doi:10.1145/1455770.1455793.
- 17 Paolo G. Giarrusso, Léo Stefanescu, Amin Timany, Lars Birkedal, and Robbert Krebbers. Scala step-by-step: soundness for DOT with step-indexed logical relations in Iris. *Proc. ACM Program. Lang.*, 4(ICFP):114:1–114:29, 2020. doi:10.1145/3408996.
- 18 James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The Java® language specification, Java SE 8 edition. <https://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.4.1>, 2022.
- 19 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, OOPSLA '12*, pages 879–896. Association for Computing Machinery, 2012. doi:10.1145/2384616.2384680.
- 20 JML reference manual: Class and interface member declarations. https://www.cs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_7.html#SEC60, 2022.
- 21 Ifaz Kabir. themapplelab / dot-public: A simpler syntactic soundness proof for dependent object types. <https://github.com/themapplelab/dot-public/tree/master/dot-simpler>.
- 22 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.
- 23 Ifaz Kabir, Yufeng Li, and Ondřej Lhoták. iDOT: a DOT calculus with object initialization. *Proc. ACM Program. Lang.*, 4(OOPSLA):208:1–208:28, 2020. doi:10.1145/3428276.

- 24 Darya Melicher, Anlun Xu, Valerie Zhao, Alex Potanin, and Jonathan Aldrich. Bounded abstract effects. *ACM Trans. Program. Lang. Syst.*, 44(1), January 2022. doi:10.1145/3492427.
- 25 David A. Naumann. Observational purity and encapsulation. In Maura Cerioli, editor, *Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3442 of *Lecture Notes in Computer Science*, pages 190–204. Springer, 2005. doi:10.1007/978-3-540-31984-9_15.
- 26 Jens Nicolay, Quentin Stiévenart, Wolfgang De Meuter, and Coen De Roover. Purity analysis for JavaScript through abstract interpretation. *Journal of Software: Evolution and Process*, 29(12), 2017. doi:10.1002/smrv.1889.
- 27 Martin Odersky, Aleksander Boruch-Gruszecki, Jonathan Immanuel Brachthäuser, Edward Lee, and Ondřej Lhoták. Safer exceptions for Scala. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Scala, SCALA 2021*, pages 1–11, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3486610.3486893.
- 28 Martin Odersky, Aleksander Boruch-Gruszecki, Edward Lee, Jonathan Brachthäuser, and Ondřej Lhoták. Scoped capabilities for polymorphic effects, 2022. arXiv:2207.03402.
- 29 David J. Pearce. JPure: A modular purity system for Java. In Jens Knoop, editor, *Compiler Construction - 20th International Conference, CC 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 6601 of *Lecture Notes in Computer Science*, pages 104–123. Springer, 2011. doi:10.1007/978-3-642-19861-8_7.
- 30 Marianna Rapoport, Ifaz Kabir, Paul He, and Ondřej Lhoták. A simple soundness proof for dependent object types. *Proc. ACM Program. Lang.*, 1(OOPSLA):46:1–46:27, 2017. doi:10.1145/3133870.
- 31 Marianna Rapoport and Ondřej Lhoták. A path to DOT: formalizing fully path-dependent types. *Proc. ACM Program. Lang.*, 3(OOPSLA):145:1–145:29, 2019. doi:10.1145/3360571.
- 32 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. ACM Press, 2017. doi:10.1145/3103111.3104036.
- 33 Tiark Rompf and Nada Amin. Type soundness for dependent object types (DOT). In Eelco Visser and Yannis Smaragdakis, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016, OOPSLA '16*, pages 624–641. ACM, 2016. doi:10.1145/2983990.2984008.
- 34 Atanas Rountev. Precise identification of side-effect-free methods in Java. In *20th International Conference on Software Maintenance (ICSM 2004), 11-17 September 2004, Chicago, IL, USA*, pages 82–91. IEEE Computer Society, 2004. doi:10.1109/ICSM.2004.1357793.
- 35 Lukas Rytz, Nada Amin, and Martin Odersky. A flow-insensitive, modular effect system for purity. In Werner Dietl, editor, *Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs, FTfJP 2013, Montpellier, France, July 1, 2013, FTfJP '13*, pages 4:1–4:7. ACM, 2013. doi:10.1145/2489804.2489808.
- 36 Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming, LFP '92*, pages 288–298, New York, NY, USA, 1992. Association for Computing Machinery. doi:10.1145/141471.141563.
- 37 Alexandru Salcianu and Martin Rinard. A combined pointer and purity analysis for Java programs. Technical report, Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory, 2004. URL: <https://dspace.mit.edu/handle/1721.1/30470>.

- 38 Alexandru Salcianu and Martin C. Rinard. Purity and side effect analysis for Java programs. In Radhia Cousot, editor, *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings*, volume 3385 of *Lecture Notes in Computer Science*, pages 199–215. Springer, 2005. doi: [10.1007/978-3-540-30579-8_14](https://doi.org/10.1007/978-3-540-30579-8_14).
- 39 Guannan Wei, Oliver Bračevac, Songlin Jia, Yuyan Bao, and Tiark Rompf. Polymorphic reachability types: Tracking freshness, aliasing, and separation in higher-order generic programs. *Proc. ACM Program. Lang.*, 8(POPL), January 2024. doi: [10.1145/3632856](https://doi.org/10.1145/3632856).
- 40 Haiying Xu, Christopher J. F. Pickett, and Clark Verbrugge. Dynamic purity analysis for Java programs. In Manuvir Das and Dan Grossman, editors, *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE’07, San Diego, California, USA, June 13-14, 2007*, pages 75–82. ACM, 2007. doi: [10.1145/1251535.1251548](https://doi.org/10.1145/1251535.1251548).

The Performance Effects of Virtual-Machine Instruction Pointer Updates

M. Anton Ertl   

TU Wien, Austria

Bernd Paysan

net2o, Munich, Germany

Abstract

How much performance do VM instruction-pointer (IP) updates cost and how much benefit do we get from optimizing them away? Two decades ago it had little effect on the hardware of the day, but on recent hardware the dependence chain of IP updates can become the critical path on processors with out-of-order execution. In particular, this happens if the VM instructions are light-weight and the application programs are loop-dominated. The present work presents several ways of reducing or eliminating the dependence chains from IP updates, either by breaking the dependence chains with the *loop* optimization or by reducing the number of IP updates (the *c* and *ci* optimizations) or their latency (the *b* optimization). Some benchmarks see speedups from these optimizations by factors > 2 on most recent cores, while other benchmarks and older cores see more modest results, often in the speedup ranges 1.1–1.3.

2012 ACM Subject Classification Software and its engineering → Virtual machines; Computer systems organization → Superscalar architectures; Software and its engineering → Interpreters

Keywords and phrases virtual machine, interpreter, out-of-order execution

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.14

Supplementary Material Collection (Source Code, Binaries, Data): <https://www.complang.tuwien.ac.at/anton/ip-updates.tar.xz> [10]

Software (Source Code): <https://git.savannah.gnu.org/cgit/gforth.git> [9]
archived at sw.h1:dir:61eb3b71325060fe2e01f5e819eb0bec959e5bf0

1 Introduction

Interpreters are a popular approach for implementing programming languages. Their benefits are simplicity of implementation, portability, and fast edit-run cycles. While they cannot compete in execution performance with JIT compilers or ahead-of-time compilers, a fast interpreter is not that far away: e.g., with the IP update optimizations of the present work, Gforth has similar performance to the SwiftForth JIT compiler and to gcc -O0 (see Section 6).

This paper uses Gforth as an example high-performance interpreter. Gforth implements a virtual machine (VM) and uses several previously published techniques for achieving high performance (see Section 2), most notably dynamic superinstructions (aka selective inlining) with replication and stack caching.

At the start of this work, every VM instruction in Gforth performed a VM instruction-pointer (IP) update [3]. It turns out that these IP updates (both the increments for ordinary instructions and the loads for taken branches) form a critical dependence path that limits the execution performance of many programs on modern processors.

We introduce a collection of optimizations for reducing these dependences: The loop optimization (*l*) breaks dependency chains in loops (Section 4.1). Optimization *c* combines the IP updates of VM instructions that do not need an up-to-date IP (Section 4.2); the immediate optimization (*i*) avoids the need for an up-to-date IP for VM instructions with immediate operands (Section 4.3); The branch optimization (*b*) optimizes VM branches by replacing loads with (lower-latency) adds (Section 4.4).

Section 2 explains the interpreter performance techniques necessary to understand the present work. Section 3 explains how data dependences influence the performance of modern processors. Section 4 describes the optimizations and shows an example of their application; the novel *loop* (Section 4.1), *immediate* (Section 4.3), and *branch* (Section 4.4) optimizations are among the main contributions of this work. Section 5 describes the measurement setup. The other main contribution of this work is in the empirical evaluation of the optimizations (Section 6). Finally, we discuss the applicability to other languages (Section 7), how to get the source code (Section 8) and related work (Section 9).

1.1 Why Gforth? Is this paper relevant for other languages?

You may wonder why we use Gforth and whether our results are relevant for other languages and their VMs.

We chose Gforth in the present work because it already implemented a number of techniques for increasing performance, in particular dynamic superinstructions and stack caching. As a result, Gforth’s VM executes so few real-machine instructions per VM instruction that the dependences formed from IP updates become a bottleneck on certain programs.

We think that our IP update optimizations are also applicable to other VMs, but it depends on the VM, its implementation, and the characteristics of programs that are run on it how big the benefits will be. For a longer discussion, see Section 7.

2 Interpreter performance techniques

This section provides an overview of the performance techniques as far as necessary for understanding the IP update optimization, with literature references.

2.1 Virtual machines

Most interpreted programming language implementations compile the source code with a simple compiler into an intermediate code that represents the source program as a sequence of instructions of a virtual machine (VM) that is designed as both an easy target for the compiler and for easy (and ideally efficient) interpreted implementation of this code. Some well-known virtual machines, such as the Java Virtual Machine [15] and WebAssembly [12] also serve as program interchange formats, but in the present paper we focus on the role of virtual machines for execution in fast interpreters.

For our running example, we use Gforth’s VM. Gforth is an implementation of the programming language Forth, a low-level (address arithmetic etc.) stack-based programming language.

Our running example is the inner loop of the siev benchmark:

```
do
  0 i c!
  dup +loop
```

We look only at the body of the loop, i.e., without the `do`. In Gforth’s VM, the body looks as follows:

```

loophead: lit
  0
  i
  c!
  dup
  (+loop)
  loophead

```

Each line occupies one machine word, and *slanted blue* lines are immediate operands of the preceding VM instruction.

An interpreter for VM code keeps a pointer to the current VM instruction (the IP) around and uses it for finding immediate operands of the VM instruction and for finding the next VM instruction. In case of a VM-level direct branch instruction like `(+loop)`, the immediate operand is the branch target and if the branch is taken, the IP is set to the value of the immediate operand.

That's all you need to understand the optimization in the paper in the abstract, but to round out the picture, the rest of this section describes what these VM instructions do.

This Forth code corresponds to the following C code:

```

do {
  *p = 0;
  p += prime;
} while (p<pend)

```

Gforth's VM is stack-based and is relatively close to the Forth source code, with the following exceptions: Gforth compiles the number 0 to the VM instruction `lit` with the immediate operand `0`, and it compiles `+loop` to the VM instruction `(+loop)` with an immediate operand: the address of the VM instruction that `(+loop)` jumps to unless it exits the loop.

`lit` pushes its immediate operand on the data stack (or stack, for short). `i` pushes the current counter of the `do...+loop` counted loop on the stack (in this loop the counter contains the address corresponding to `p` in the C fragment). `c!` (pronounced “c-store”) stores the second item on the stack to the byte pointed to by the address on the top-of-stack (TOS), popping both stack items. `dup` pushes another copy of the current top-of-stack value on the stack; this value corresponds to `prime` in the C program.

`(+loop)` pops the top-of-stack and adds it to the loop counter and checks for loop termination.¹ If another iteration is merited, `(+loop)` performs a VM-level jump to `loophead`.

2.2 Switch dispatch

A common way to implement an interpreter in C is to use a big switch statement along the lines of:

¹ As you will see in the assembly code later, this check is more complex than one would expect from the C code. The reason is that `+loop` is specified to support circular arithmetic and both positive and negative increments, which complicates the termination condition. For details see <https://forth-standard.org/standard/core/PlusLOOP>.

14.4 The Performance Effects of Virtual-Machine Instruction Pointer Updates

```

for (;;) {
    switch (*ip) {
        case dup: dsp[0] = tos; dsp--; ip++; break;
        case lit:  dsp[0] = tos; dsp--; tos = ip[1]; ip+=2; break;
        ....
    }
}

```

In this example the data stack is represented by having the top-of-stack in a local variable `tos`, and the remainder of the data stack is in memory, and the local variable `dsp` points to where the top-of-stack would reside if it were in memory. IP is also kept in a local variable `ip`. We will use the same names for registers in assembly code shown below.

This scheme has a relatively high overhead of getting from one VM instruction implementation to the next. For `lit` with switch dispatch `gcc -O2` produces the following code for RISC-V (the destination register (if any) is leftmost):

```

.L2:          #switch code
    ld      a4,0(ip)   # a4=*ip
    slli   a5,a4,2     # a5=a4*4 #for indexing
    add    a5,a6,a5     # a5=a6+a5 #table start in a6
    bgtu  a4,a7,.L17   # if a4>a7 goto default #bounds check
    lw     a2,0(a5)     # a2 = *a5 #load from table
    jr    a2             # indirect branch to a2

.L6:          #lit code
    sd      tos,0(dsp)  # dsp[0] = tos
    ld      tos,8(ip)   # tos = ip[1]
    addi   dsp,dsp,-8   # dsp--
    addi   ip,ip,16     # ip += 2
    j     .L2            # back to switch code

```

Figure 1 shows the data structures involved in switch dispatch. The VM instructions are represented as integers that are used as indexes into the switch table. We use 8-byte VM-code slots for the code above.

The *payload* consists of only 3 RISC-V instructions in this case, whereas the dispatch overhead is 8 instructions.

Gforth has never implemented switch dispatch, and instead went directly for threaded code.

2.3 Threaded code

Threaded code [1] reduces the dispatch overhead by representing each VM instruction directly as the address of the machine code that implements it. This means that each instruction occupies one machine word (8 bytes on a 64-bit machine) and immediate operands are usually represented by one or more machine words. This concept results in the following code for `lit`:

```

sd      tos,0(dsp) # dsp[0] = tos
ld      tos,0(ip)   # tos = ip[0]
addi   dsp,dsp,-8 # dsp--
addi   ip,ip,16     # ip += 2
ld      a4,-8(ip)  # a4 = ip[-1] #address of next VM inst
jr    a4            # jump to next VM inst

```

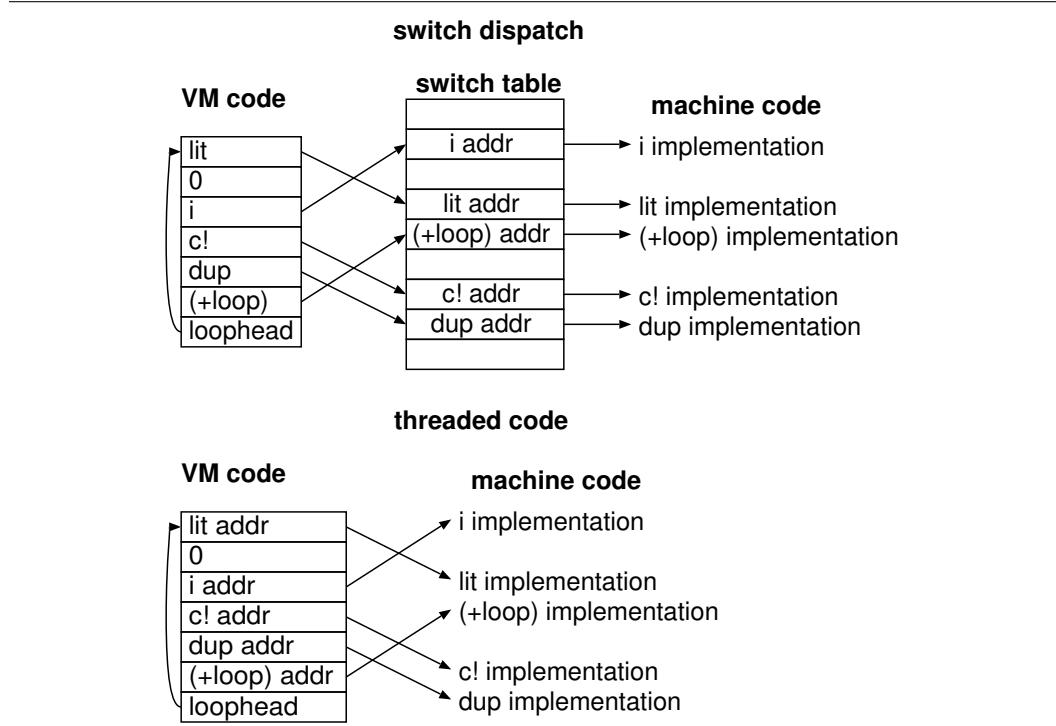


Figure 1 Switch dispatch vs. threaded code.

The dispatch code is inlined here and consists of 3 RISC-V instructions.

Figure 1 shows how the two schemes get from the VM code to the corresponding machine code. In both schemes ip points to the VM code, and immediate operands are accessed through ip. VM control flow is performed by setting ip to something other than the next VM instruction.

One practical consideration is how to implement threaded code in an architecture independent language. Fortunately it is possible by using the labels-as-value extension of GNU C, which has been implemented by at least gcc, clang, tcc, and icc.

2.4 Selective inlining and dynamic superinstructions

One can eliminate more of the dispatch: While generating VM code, copy (real-)machine code snippets from the interpreter to a separate memory area, thus concatenating these snippets (Fig. 2); the threaded-code addresses then point to this newly generated real-machine code rather than the originals as in normal threaded code.

This technique has first been outlined as *memcpy method* by Rossi and Sivalingham [20], and later explored in depth as *selective inlining* by Piumarta and Riccardi [17]. Ertl and Gregg combined it with replication [4] for better branch prediction. They call the result of the concatenation *dynamic superinstructions*, because, like static superinstructions [18, 8, 2] they combine a sequence of n VM instructions with n dispatches into something with only one dispatch.

In our example (Fig. 2), each VM instruction except the last one ((+loop), which is a VM-level branch) just continues with the next one, so the machine code of the next one can be concatenated to the machine code of the dynamic superinstruction. The (+loop) may

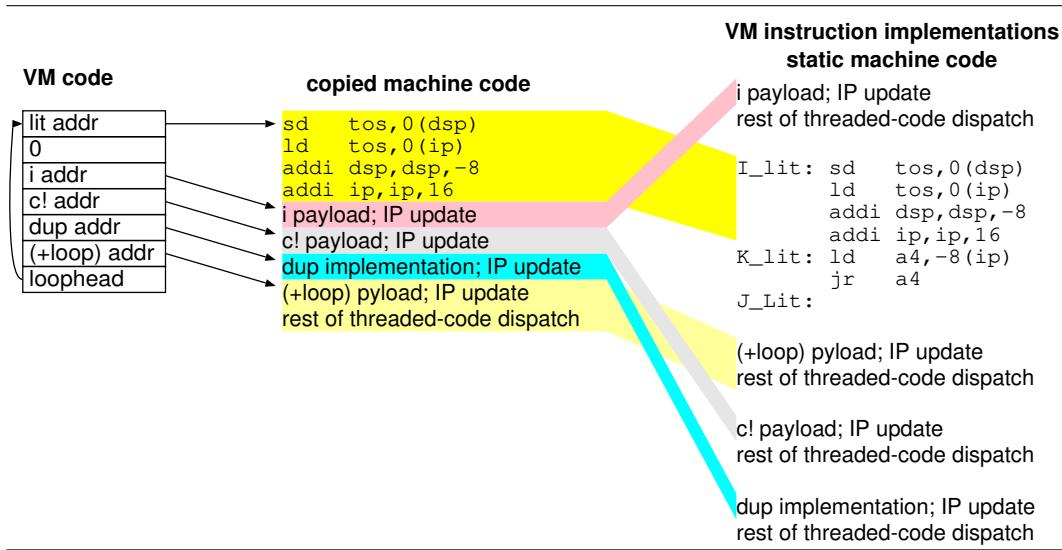


Figure 2 Concatenating machine-code snippets to further reduce the dispatch overhead.

set `ip` to something other than the next instruction, so for `(+loop)` and other control-flow VM instructions the whole code including the rest of a threaded-code dispatch is appended in order for the control-flow change to take effect at run-time; such an instruction therefore ends a dynamic superinstruction.

Gforth copies the machine code snippets at the same time as when it generates VM code for newly compiled source code. Gforth does not save the machine code in its images, so its image loader copies the machine code snippets for the VM code it loads.

This technique has provided a big performance boost to Gforth across many different CPUs, typically by a factor of 2 over threaded code (see Fig. 10). One may balk at the prospect of directly manipulating machine code, but the advantage of starting with an interpreter is that Gforth can always fall back to threaded code if conditions seem adverse (and this normally works automatically).

One may wonder if the result is not already a JIT compiler, and in certain respects it is. But for the language implementor it is an extension of a threaded-code interpreter: Each implementation of a VM instruction just gets labels before and after the “rest of threaded-code dispatch” part, and when a VM instruction is generated, it also copies the memory containing the machine code for the VM instruction (using the labels to know the boundaries), and lets the threaded-code word point to the copy (instead of the original). The only amount of machine-specific code are a few lines to synchronize the I-cache to the D-cache, and GNU C provides `__builtin__clear_cache` for that purpose. And when the conditions for dynamic code generation are not met, the system just falls back to plain threaded code, overall or on a per-VM-instruction basis (e.g., for code that contains a relative reference to an address outside the code snippet at hand). By contrast, a typical JIT compiler needs much more machine-specific work.

2.5 Multi-representation stack caching

The Gforth baseline also uses an optimization called multi-representation stack caching. This optimization reduces only the machine instructions in the payload, so you only need to read this section if you want to understand the payload of our running example, too.

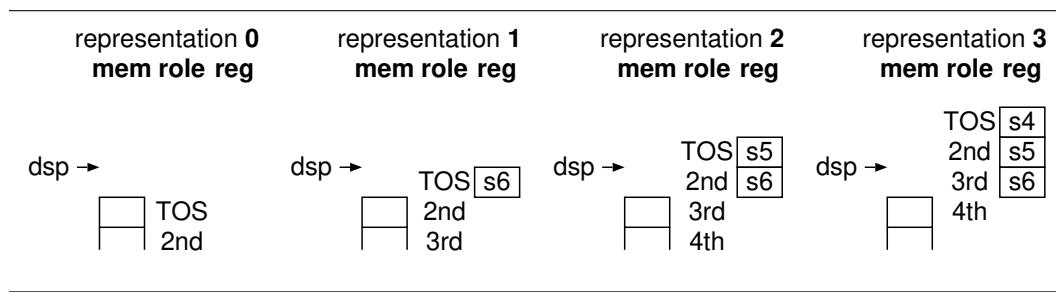


Figure 3 Four data-stack representations used by Gforth on RISC-V.

Figure 3 shows different representations of the data stack. Representation 0 keeps 0 stack items in registers, i.e., all stack items in memory. A representation with all stack items in memory is often seen in the literature (usually with the stack pointer pointing to the top-of-stack, but that is just a difference in the offsets used for the memory accesses).

The examples shown earlier use representation 1, and this is also used by Gforth when it falls back to threaded code. The advantage of this representation can be seen for `dup` which does one load and two stores with representation 0, but just one store with representation 1.

By switching between representations Gforth further reduces the stack handling effort. E.g., our running example starts in representation 1 (Gforth always uses this at the start of a basic block) with the VM instruction `lit`. By choosing the `lit` implementation that ends in representation 2 (i.e., `lit 1 → 2`), the payload of `lit` in this case is reduced to

`ld s5,8(ip)`

The old top-of-stack stays in `s6` (and becomes the 2nd stack element), and the new top-of-stack is pushed by setting `s5` (the new top-of-stack) to the immediate operand.

This eliminates a memory access to the data stack as well as an update of `dsp`. If you take a closer look at Fig. 4, you do not find any memory access to the data stack nor any data-stack pointer update, so in this case data-stack caching works perfectly.

Ertl and Gregg [5, 7] discuss multi-representation stack caching in more detail.

3 Understanding performance

This section describes how program characteristics influence the performance on processors with out-of-order (OoO) execution, and in particular, it discusses the role of instruction pointer updates in interpreters with dynamic superinstructions. OoO processors have dominated general-purpose computers in this century, and are now advancing towards smaller systems. E.g., the Raspberry Pi switched to OoO cores with the Raspberry Pi 4 in 2019 and the Compute Module 4 in 2020.

3.1 ... on modern CPUs ...

Starting from an empty pipeline, the front end of an OoO processor fetches and decodes instructions as directed by the branch predictor, possibly running far ahead of execution. An instruction is then executed as soon as all its inputs are available and an appropriate functional unit is available.

If a branch is mispredicted, fetch, decode, and execution at first continue along the (mis-)predicted path, but the results are not committed. When the correct direction or branch target is determined by executing the appropriate conditional or indirect branch instruction, the front end is redirected to fetch, decode and eventually execute from the correct path.

This description indicates the ways in which program characteristics influence performance:

As long as mispredictions are rare, if there are enough independent instructions, execution will be limited by the resources, either by the program needing too many of a particular functional unit (e.g., a matrix multiply program will exercise load and store units and the FP multiply-add a lot), or by the width of the decoder and/or the retirement unit.

On the other hand, if there are lots of dependences between instructions and the processor offers enough resources, the dependences will determine the performance: an instruction that depends on another instruction i on the critical dependence path will wait in the processor's buffers until i produces a result. After prefetching for a while, all these not-yet ready instructions will fill the processor's buffers and the processor's front end has to wait until more buffers become ready by finishing an instruction on the critical path.

In the branch misprediction case, the misprediction penalty is influenced by the kind of dependences between instructions: If there is a short dependence path to the predicted branch instruction, the misprediction can be resolved early. However, if the mispredicted branch depends on an instruction in the critical dependence path, the misprediction will not be discovered until the instructions leading to the branch have been executed; only then can the correct path be fetched and decoded, so such a misprediction incurs a bigger misprediction penalty. By contrast, in case of a correct prediction, the long latency until the prediction is confirmed does not hurt, except for occupying some buffers for longer.

3.2 ... in fast interpreters

In an interpreter, there is the resource consumption and dependences inherent in the interpreted program (i.e., also present if the program is compiled to real-machine code), but there is also the overhead of the interpreter:

In particular, every VM interpreter updates the VM instruction pointer (IP), in order to access immediate VM data through it, and to access the next VM instruction (or next dynamic superinstruction). In straight-line code, this results in one addition per executed VM instruction, with a latency of one cycle on most processors. For a VM-level absolute branch (as used in Gforth), the new VM instruction pointer has to be loaded, with a latency of 3–5 cycles on recent OoO processors; if the VM-level branch is relative, the loaded value has to be added to the instruction pointer, costing an additional cycle.

A VM-level return instruction breaks the IP dependence chain of the callee, because it loads the new VM instruction pointer from the saved return address. This continues the dependence chain of the caller, but the callee's chain of IP updates ends with the return.

VM interpreters also have other overheads: A stack-based VM like that of Gforth has dependence chains through stack-pointer updates, and these dependence chains are not broken by returns. Moreover, they keep most stack items in memory with the resulting store-to-load latency: 4–7 cycles in many processors, but 0 cycles in several recent processors like Zen 3 and Tiger Lake.² However, stack caching (see Section 2.5) reduces these overheads substantially.

² <https://www.complang.tuwien.ac.at/anton/memdep/>

For a register-based VM, the VM register accesses are usually implemented through real-machine memory accesses, which increases the resource consumption substantially. On older processors there is also the latency cost of store-to-load forwarding, but the significance of this cost depends on the dependence patterns of the interpreted program.

Previous work did not consider VM instruction-pointer updates to have much effect. Ertl and Gregg [4] wrote:

One thing that we have not implemented is eliminating the increments of the VM instruction pointers along with the rest of the instruction dispatch in dynamic superinstructions. However, by using static superinstructions in addition dynamic superinstructions and replication we also reduce these increments (in addition to other optimizations); looking at the results from that, eliminating only the increments probably does not have much effect.

For a long time our thinking was that other dependencies would dominate over VM instruction-pointer updates, and that, with processors becoming wider (being able to execute more instructions per cycle), instruction-pointer updates would become even less relevant. However, for a number of benchmarks this is wrong (see Section 6).

4 Instruction-pointer update optimization

This section discusses four mostly independent optimizations. We implemented these optimizations in Gforth, and discuss them in this context, but they can also be applied to implementations of other languages.

4.1 Loops

This optimization breaks the IP dependence chains on loop-back edges. In typical VM instruction sets, the loop-back branch takes the target address as an immediate operand (e.g., in Fig. 4 the immediate operand **loophead** following the VM branch instruction (`+loop`)).

With the *loop* optimization, the loop-back address is stored on the return stack on entering the loop, and the loop-back branch then takes its address from there (the **bold green** instruction in Fig. 4). Because it does not need to access the VM instruction pointer to do that, this breaks the dependence chain.

In Forth the return stack is a stack that contains return addresses and counted-loop parameters. In general, the loop-back address can be stored on any stack or in a VM register; the important part for the *loop* optimization is that this address must be readable by the loop-back instruction without requiring an IP access, so one cannot use a VM register whose number is given as immediate operand.

Unfortunately, the design of Forth makes it difficult to apply this optimization to general loops, so we only apply it to counted loops in the present work.

However, if you are designing a virtual machine for a programming language, it may be worthwhile to design it in a way that makes it possible to store the loop-back address somewhere on entry to the loop, and to load it from there on the loop-back branch without accessing the IP.

The loop optimization has very little effect on the instruction count and other dependences, and can therefore be used to see the performance effect of breaking the IP dependence chains independent of, e.g., the effect of reducing the number of executed instructions. In Fig. 9 we see speedups by a factor of 2 on some benchmarks, showing that the IP-update dependence chain really is the bottleneck for these benchmarks.

14:10 The Performance Effects of Virtual-Machine Instruction Pointer Updates

VM code		<i>unoptimized</i>	<i>l</i>	<i>c</i>	<i>ci</i>	<i>cib</i>
lit	1 → 2	<i>addi ip,ip,16</i>	<i>addi ip,ip,16</i>	<i>addi ip,ip,16</i>		
<i>O</i>		ld s5,-8(ip)	ld s5,-8(ip)	ld s5,-8(ip)	ld s5,8(ip)	ld s5,8(ip)
i	2 → 3	<i>addi ip,ip,8</i>	<i>addi ip,ip,8</i>			
		ld s4,0(rp)	ld s4,0(rp)	ld s4,0(rp)	ld s4,0(rp)	ld s4,0(rp)
c!	3 → 1	<i>addi ip,ip,8</i>	<i>addi ip,ip,8</i>			
		sb s5,0(s4)	sb s5,0(s4)	sb s5,0(s4)	sb s5,0(s4)	sb s5,0(s4)
dup	1 → 2	<i>addi ip,ip,8</i>	<i>addi ip,ip,8</i>			
		mv s5,s6	mv s5,s6	mv s5,s6	mv s5,s6	mv s5,s6
(+loop) <i>loophead</i>	2 → 1	<i>addi ip,ip,16</i>	<i>addi ip,ip,8</i>	<i>addi ip,ip,40</i>	<i>addi ip,ip,56</i>	
		ld a5,0(rp)	ld a5,0(rp)	ld a5,0(rp)	ld a5,0(rp)	ld a5,0(rp)
		ld a4,8(rp)	ld a4,8(rp)	ld a4,8(rp)	ld a4,8(rp)	ld a4,8(rp)
		<i>ld a2,-8(ip)</i>		<i>ld a2,-8(ip)</i>	<i>ld a2,-8(ip)</i>	
		add a3,s5,a5	add a3,s5,a5	add a3,s5,a5	add a3,s5,a5	add a3,s5,a5
		sub a4,a5,a4	sub a4,a5,a4	sub a4,a5,a4	sub a4,a5,a4	sub a4,a5,a4
		add a4,s5,a5	add a4,s5,a5	add a4,s5,a5	add a4,s5,a5	add a4,s5,a5
		xor a5,a4,a5	xor a5,a4,a5	xor a5,a4,a5	xor a5,a4,a5	xor a5,a4,a5
		xor a5,s5,a5	xor a5,s5,a5	xor a5,s5,a5	xor a5,s5,a5	xor a5,s5,a5
		and a4,a5,a4	and a4,a5,a4	and a4,a5,a4	and a4,a5,a4	and a4,a5,a4
		sd a3,0(rp)	blt a5,zero,x	sd a3,0(rp)	sd a3,0(rp)	sd a3,0(rp)
		blt a5,zero,x	<i>ld ip,16(rp)</i>	blt a5,zero,x	blt a5,zero,x	blt a5,zero,x
		ld a5,0(a2)	sd a3,0(rp)	ld a5,0(a2)	ld a5,0(a2)	ld a5,0(ip)
		<i>mv ip,a2</i>	ld a5,0(ip)	<i>mv ip,a2</i>	<i>mv ip,a2</i>	
		jr a5	jr a5	jr a5	jr a5	jr a5
		x:	x:	x:	x:	x:

Figure 4 The inner loop of the benchmark `siev` in Gforth’s VM code, and the corresponding RISC-V code produced by Gforth without optimization and with various IP-update optimizations: *l* optimizes loops, *c* combines IP updates, *i* optimizes immediate operands, *b* optimizes branches. 1 → 2 etc. indicates a stack representation change (see Section 2.5). For instructions with destination registers, the destination is leftmost. The instruction that starts a new IP dependence chain (in the loop) is **bold green**. Instructions that continue IP update dependence chains are **slanted red**. Some register names have been changed for ease of understanding: ip is the VM instruction pointer, rp is the return-stack pointer. s6, s5, s4 contain stack elements (see Fig. 3).

While it is possible to combine this optimization with the others, we think that the combination of the others is effective enough in reducing the IP dependence chain, and that adding the *loop* optimization would not help once the other optimizations are performed. However, we consider the *loop* optimization to be an alternative that requires less effort.

You can see the result in column *l* of Fig. 4. The decisive difference is that the ***ld a2,-8(ip)*** in (+loop) in *unoptimized*, *c*, *ci* loads the branch target from VM code using the IP, while the ***ld ip,16(rp)*** loads the branch target from the return stack (using rp).

We implemented a prototype of this optimization in Gforth by adding 89 lines.

4.2 Combining instruction-pointer updates

There are VM instructions where the payload of the implementation does not read the IP and therefore does not need an up-to-date IP. In our running example `i`, `c!` and `dup` do not need an up-to-date IP.

Therefore the IP update can be left away. When there is finally a reason for an up-to-date IP, all the updates can be combined into one addition of a larger constant.

Columns *unoptimized*³ and *c* of Fig. 4 illustrate this. In *unoptimized* every VM instruction has its own IP update; in *c*, `lit` has an IP update, because it loads its immediate operand 0 in the VM code through `ip`. The next three VM instructions `i`, `c!` and `dup` don't need an up-to-date IP, so `c` eliminates their IP updates. Finally, `(+loop)` needs an up-to-date IP in order to load its immediate operand `loophead` (the loop-back address) from the VM code, so Gforth's compiler inserts an IP update by 40 covering all VM instructions `i...(+loop)` (inclusive), the same as the sum of the corresponding IP updates in *unoptimized*.

The optimization itself is trivial: The code generator just keeps track of where IP actually points to, and when an up-to-date IP is needed, it inserts the appropriate update.

One not quite trivial part, however, is: When is an up-to-date IP needed?

Superblock end: The next VM instruction is the target of a VM jump. Because the IP may be used afterwards, we have to synchronize the IPs coming from different paths at this point, and we do it by letting it point to the first VM instruction in the new superblock.

Calls: VM instructions like `call` and `execute` (an indirect call) also require an up-to-date IP: calls save the IP (which points to the next instruction at that point) as return address, and after returning execution continues at that address. The routine invoked by `execute` finishes with a threaded-code dispatch, which needs an up-to-date IP.

Non-relocatable VM instruction: When the machine code for a VM instruction is not relocatable (typically because there is a call to a C function in the machine code), this code cannot be used in a dynamic superinstruction. Instead, this code is called through a threaded-code dispatch (which uses IP), and this code then updates the IP and makes another threaded-code dispatch for continuing execution after this VM instruction.

Immediate operands: The IP is used when accessing immediate operands of VM instructions.

One particular case of this optimization is VM instructions like Gforth's `;s` which returns from a definition. It does not need an up-to-date IP beforehand, and it branches elsewhere (returning to the caller at the VM level), so there is no need to update the IP afterwards, and we suppress such an update.

The other not quite trivial part is how to generate the machine code in the dynamic superinstruction framework for which the actual machine code that is copied around is just an opaque code snippet.

The first question is how to separate the IP update that is part of every VM instruction implementation (as part of the threaded-code dispatch) so that we can copy the machine code without the IP update.

Because the IP needs to be up-to-date in front of some VM instructions, we put the IP update at the start of each VM-instruction implementation, resulting in the following template:

```
I_inst:
    update ip
L_inst:
    non-dispatch code
K_inst:
    rest of dispatch
J_inst:
```

³ This column shows the code for a Gforth version that includes ip-update optimizations but has them disabled; this means that it uses the same register allocation and instruction schedule as the various optimized variants, which makes it easy to compare with the other columns.

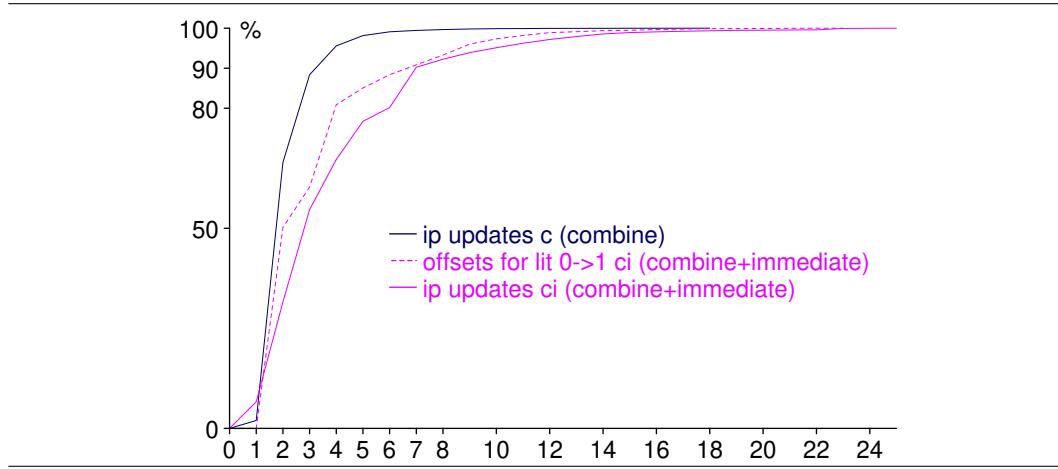


Figure 5 Proportion of IP updates or `lit 0->1` offsets with distances less than a given number of machine words in Gforth’s image (static counts).

If the resulting code is relocatable and code for the VM instruction *inst* should be generated, the code generator first generates an appropriate IP update if necessary (but normally does not use the update between `I_inst` and `L_inst` for that). Then it copies the code between `L_inst` and `K_inst`.

The code generator needs code snippets for different amounts of IP updating, because it cannot just patch a constant into a template for IP-updating (the code generator does not know anything about the internal structure of the machine code). Instead, we added code snippets for IP updates for a range of values (by 1–23 machine words) to the C source code of Gforth, and the code generator selects the right one, or (for IP updates > 23 words) generates a sequence of IP updates.

Figure 5 shows that if we have IP update code snippets for updates by 1–6 machine words, 99% of the cases statically occurring in the Gforth image can be performed with one instruction, so for Gforth limiting the IP upates to this range would be good enough as long as VM instructions with immediate operands (Section 4.3) are not optimized as well.

These data are somewhat specific to the Gforth VM, so if you want to minimize the number of IP update code snippets, you should do your own measurements. As Fig. 5 shows, a major reason for IP updates is VM instructions with immediate operands. VMs that use local-variable accesses more than Gforth and specify the local with an immediate operand will have shorter sequences between IP updates, which makes *c* alone less beneficial, but means that even fewer IP update code snippets cover nearly all occurring distances with one IP update.

4.3 Immediate operands

VM instructions with immediate operands are relatively frequent. We can eliminate this reason for requiring an up-to-date IP in most cases: We introduce additional variants of the most frequent VM instructions with immediate operands.⁴ These additional variants access their immediate operand at an offset (1–23 machine words in our experimental implementation) from where their base variant accesses the immediate operand, thus allowing the actual IP to point 1–23 machine words in front of to up-to-date IP.

⁴ `lit`, `call`, `?branch`, `lit@`, `branch`, `(loop)`, `lit-perform`, `lit+`, `does-xt`

When the code generator has to compile such a VM instruction, if the difference between the actual and the up-to-date IP is within the offset range of the variants, the code generator copies the code of the appropriate variant, and no IP update needs to be generated.

In Fig. 4, column *ci* shows how combining IP updates is enhanced by this immediate-operand optimization: The first VM instruction is `lit`, and in *c* it needs an IP update; in *ci*, a variant of `lit` that accesses its immediate operand at `ip+8` instead of `ip-8` (an offset of 2 machine words) is used, so there is no need to update IP.

However, `(+loop)` is a VM instruction that does not have such variants, so the code generator updates the IP at the start of `(+loop)`.

While it is not obvious from this example, this extension contributes a lot to the effectiveness of combining IP updates: In the Gforth image, the number of static IP updates is reduced by a factor of 5; the dynamic reduction in our benchmarks usually a factor < 2 compared to the reduction from *c* alone (see Fig. 8).

Figure 5 shows that IP updates by 1–16 machine words are sufficient for performing (without resorting to sequences of adds) 99% of the remaining IP updates statically occurring in the Gforth image. It also shows that for the most frequent VM instruction with a literal, `lit` in its stack caching variant $0 \rightarrow 1$, 99% of the IP offsets are in the range 2–13 (machine words).

The relevance of these numbers is as follows: The compilation time of the VM implementation increases with the number of VM instruction implementation variants, so we only want to add additional variants when a benefit is expected. This is particularly relevant for instructions with immediate operands, because there are a number of them, and stack caching multiplies the numbers.

E.g., we selected only 9 VM instructions with immediate operands; stack caching increases this to 15 variants, and having 24 subvariants with different offsets for each variant results in a total of 360 implementations of these 9 VM instructions. We did not use additional variants for other VM instructions with immediate operands (e.g., `(+loop)`) to avoid increasing the compilation time of the interpreter too much. For the same budget of 360 implementations, it might have been a little better to use a smaller offset range and to have offset-variants of more VM instructions.

Another way to deal with this problem is to eliminate immediate operands by introducing versions of VM instructions for specific immediate operands. E.g., Gforth has a general VM instruction `@local#` with an immediate operand *n* for pushing the value of local variable *n* onto the stack, but it also has `@local0`, which fetches the local variable 0 without needing an immediate operand. To increase the benefits from IP update optimization, we added more such variants to Gforth.⁵

These optimizations also shift the balance in VM design towards splitting one VM instruction into several, especially if it means that an unoptimized VM instruction with a literal operand can be replaced with an optimized one. E.g., we have replaced the general case of `@local# n` (i.e., cases not covered by specialized variants like `@local0`) in Gforth by the sequence `lit n; @localn` where `@localn` takes *n* from a register representing the top of the data stack (pushed there by `lit`). The resulting code is often better than for `@local#`:

⁵ In the mainline, not in the variants used for the empirical results of the present work. The benchmarks used for the present work don't use local variables much, so we don't expect that this would make a significant difference.

unsplit			split		
@local#	0 → 1	<i>addi ip,ip,88</i>	lit	0 → 1	
64		ld a5,-8(ip)	64		ld s6,80(ip)
		add a5,a5,lp	@localn	1 → 1	add a5,s6,lp
		ld s6,0(a5)			ld s6,0(a5)

In this code *lp* is a register containing Gforth's locals pointer.

4.4 Branches

When executing VM instructions, every taken VM branch that loads the target address from the VM code (such as `(+loop)` in Fig. 4) performs an IP-dependent load, and thus extends the IP dependence chain with the load latency (3–5 cycles on modern processors). Even with the *ci* optimizations, these loads can mean that IP updates are still the critical dependence path in branch-heavy code like the `siev` benchmark.

However, branches are often to nearby targets, which inspires the following idea: If the target is nearby, set the IP to the target, and then execute a branch-to-IP variant of the branch; i.e., if the branch is taken, it just needs to perform a threaded-code dispatch to branch to where the IP currently points to. If the branch is not taken, execution just continues after the branch, taking the changed IP into account.

To implement this, we have extended the code snippets for updating the IP to increment the IP by -24–23 machine words. Only one branch-to-IP variant is needed for each branch, so we implemented this additional variant for all branches where the ordinary variant just takes the target address as immediate operand; there are branches in Gforth with an additional immediate operand, and we cannot apply this optimization to those branches; fortunately, they are rarely used.

You can see an example in column *cib* in Fig. 4. Thanks to the *ci* part of the optimization, there is no IP update for the *lit*, so when Gforth's code generator reaches the `(+loop)`, the actual IP is still at the start of the loop. The code generator determines that the target is nearby, and proceeds to insert an IP update for setting IP to the branch target. Because the actual IP already points to the target location, the IP update would be by 0 bytes, and no code is generated for that, an ideal outcome; in the general case you would see one or more IP update instructions at this point. Next, the code generator appends the code of the `(+loop)` variant for the branch optimization; note that this code does not contain the instructions *ld a2,-8(ip)* and *mv ip,a2* for modifying the IP; it expects that the IP already contains the right value for taking the branch.

In our experiments, we considered the target to be nearby, if it can be reached with one IP update for conditional forward branches, or if it can be reached with three IP updates for unconditional branches and conditional backwards branches. This assumes that backwards branches are usually taken, and also takes into consideration that on the fall-through path IP update for the branch might require a followup correction.

We did this for the following reasons: For unconditional branches, three IP updates have a smaller or the same latency as a load. In case of conditional branches, a backwards branch is a loop branch and therefore probably taken.

For the conditional forward branch, a classical rule-of-thumb says that not-taken is more likely. If we use the original branch instruction instead of the branch-to-IP variant, the not-taken path may work without IP update; with the branch-to-IP variant, we incur the IP update for setting the target in either case, and in the not-taken case we may need another IP update because of an instruction with an immediate operand in the code before the branch target. One could reduce the latter cost by introducing variants of instructions with immediate operands with negative offsets, but that also has its costs.

Another idea that we have not implemented (yet) is to have IP update variants with larger granularity. E.g. have IP update variants for $-16, -15, -14, \dots, 14, 15$ machine words and then $-272, \dots, -80, -48, 47, 79, 111, \dots, 271$ machine words. This would allow to compose IP updates by $-288\dots286$ machine words by concatenating two code snippets (typically with one instruction each) using only 47 IP-update variants (the same number currently used in Gforth).

We do not present empirical data for branch distances, because they depend strongly on the programming language usage (large or small routines), the VM design (e.g, already the splitting of VM instructions discussed in Section 4.3 changes the distances), and on compiler features such as tail call optimization, inlining or jump-to-jump optimization. So you will have to do your own measurements to see the distribution of distances for your VM.

For Gforth, Fig. 9 shows speedups from *cb* over *c* or from *cib* over *ci* on most benchmarks (exceptions: brainless, cd16sim, sha512), so even the $-24\dots23$ machine-word range of IP-update variants provides some benefit for this VM.

We implemented *cib* in Gforth by inserting 864 lines and deleting 316 lines.

5 Evaluation setup

5.1 Systems

We present measurements for the versions described in Section 4. As *baseline* we use a Gforth version without any IP-update optimization work. We branched a variant from that that contains only the loop optimization, and a variant that contains all new optimizations developed in the present work, selectable individually (however, the loop optimization does not work with *cib* at the moment). The Gforth variants we measured are:

baseline The Gforth version we started from. This is the numerator in the factors shown in the speedup and instruction factor graphs. The variants/system for the specific bar is the denominator.

unoptimized The version that contains all optimizations developed in the present work, but with the optimizations turned off. While in the baseline the IP update of a VM instruction is anywhere in its code, the IP update is at the start in *unoptimized* (so the IP-update optimizations can eliminate it or replace it). We show this variant in some figures to see whether the code changes had some additional effect (and to isolate this effect, if any).

baseline+loop opt This variant adds VM instruction variants for the loop optimization (see Section 4.1) and uses these for counted loops instead of the variant that loads the branch target from the VM code.

unopt+loop opt This uses the same executable as the *unoptimized* variant, but for counted loops it uses the VM instructions that perform the loop optimization.

c: combine IP updates This uses the same executable as *unoptimized*, but enables combining IP updates (Section 4.2).

ci: c+immediate opt Like *c*, but also enables the optimization of VM instructions with immediate operands (Section 4.3).

cb: c+branch opt Like *c*, but also enables the optimization of short VM branches (Section 4.2).

cib: ci+branch opt Like *ci*, but also enables the optimization of short VM branches.

Program	Author	Description	Lines	Characteristics
bench-gc	Anton Ertl	Garbage Collector	1155	calls
brainless	David Kuehling	Chess	3648	calls, app
cd16sim	Brad Eckert	CPU emulator	937	calls, app
fcp	Ian Osgood	Chess	2046	calls, app
lexex	Gerry Jackson	Scanner Generator	3655	calls, app
siev	Gilbreath/Paysan	Count primes	25	counted loops
bubble	Hennessy/Fraeman	Sort	74	counted loops, cond. br.
matrix	Hennessy/Fraeman	Integer matrix multiply	57	counted loops
fib	Anton Ertl	Recursion	14	calls, cond. branch
fft-bench	Bernd Paysan	Fast Fourier transform	106	calls in counted loop
pentomino	Bruce Hoyt	Puzzle	516	conditional branches
sha512	Marcel Hendrix	Cryptography	538	counted loops, huge body

Figure 6 Benchmark programs used.

Evaluating b alone would also have been interesting, but we left it away for time and space reasons. However, you can see the effect of b by comparing the results of c with cb and of ci with cib .

In addition, for Fig. 10 we compare with the following systems/compilers.

PFE is an interpreted Forth system written in C that uses one C function per VM instruction implementation. PFE is designed to rely on explicit register allocation (a GCC extension) for performance, but unfortunately, for AMD64 no explicit register definitions have been added yet. We use PFE-0.33.71.

Gforth threaded code only This is the baseline Gforth with the option `--no-dynamic`, which means that it falls back to using plain threaded code (Section 2.3); this option also disables stack caching.

SwiftForth, VFX Forth Two commercial Forth systems with JIT compilers. We measured SwiftForth x64-Linux 4.0.0-RC87 and VFX Forth 64 5.43.

gcc-12 Various optimization options for GCC 12.2. Manually written C code for four of the benchmarks is available and was used for generating these results. The C programs were linked statically so that the binaries could also run on machines with older glibc implementations. For gcc the results do not include the compile time (unlike for the Forth systems).

We compiled the three Gforth branches with gcc-12.2 on Debian 12 for AMD64 and with gcc-10.2 on Debian 11 on ARM A64 and statically linked them so they would run on the other platforms we used. All variants use stack caching with 0–3 registers.

5.2 Benchmarks

We use the benchmarks shown in Fig. 6. The first five are from the appbench suite of Forth benchmarks⁶; they are substantial programs and therefore are probably more representative of significant Forth applications and idiomatic Forth usage than the other benchmarks.

The next five are small benchmarks that come with Gforth: **siev** is based on the Byte Sieve by Gilbreath, but we use Bernd Paysan’s Forth version and Al Aburto’s C version.

⁶ <https://www.complang.tuwien.ac.at/forth/appbench-1.3.zip>

μ Architecture	Architecture	Family	CPU	year
K8	AMD64	AMD P	Athlon X2 4600+	2005
Zen3	AMD64	AMD P	Ryzen 7 5800X	2021
Penryn	AMD64	Intel P	Xeon E5460	2007
Nehalem	AMD64	Intel P	Xeon X3460	2009
Sandy Bridge	AMD64	Intel P	Xeon E3-1220	2011
Haswell	AMD64	Intel P	Core i7-4790K	2014
Skylake	AMD64	Intel P	Core i5-6600K	2015
Rocket Lake	AMD64	Intel P	Xeon W-1370P	2021
Tiger Lake	AMD64	Intel P	Core i5-1135G7	2021
Golden Cove	AMD64	Intel P	Core i3-1315U	2023
Silvermont	AMD64	Intel E	Celeron J1900	2013
Goldmont	AMD64	Intel E	Celeron J3455	2016
Goldmont+	AMD64	Intel E	Celeron J4105	2017
Tremont	AMD64	Intel E	Celeron N4500	2021
Gracemont	AMD64	Intel E	Core i3-1315U	2023
Firestorm	ARM A64	Apple P	M1	2020

Figure 7 Microarchitectures measured and shown in Section 6. The year shows when the CPU we measured was released. Some of the microarchitectures were released earlier in different CPUs. “P” stands for performance core, “E” for (power or die area) efficiency core.

bubble and **matrix** are based on Hennessy’s Stanford integer benchmarks (in C), and have been translated to Forth by Marty Fraeman. Four of these benchmarks are available in Forth and C in <http://www.complang.tuwien.ac.at/forth/bench.zip>.

Pentomino and sha512 were included because they exhibit unusual performance characteristics (for Forth programs): They both spend much of their time in long definitions, with many branches for pentomino, and straight-line code for sha512.

Idiomatic Forth code calls many short routines, as exhibited in the appbench programs and in fft-bench. So the results for these programs may also be representative for other programming languages where call-heavy programs are idiomatic and implementations that neither inline nor tail-call-optimize. On the other hand, the results for the programs dominated by counted loops may be more representative for programs in Algol-family languages and for systems that tail-call optimize or inline.

5.3 Hardware

We have measured a variety of different microarchitectures and show results for them. Figure 7 gives information about what the code names we use for the microarchitectures mean.

5.4 Measurements

Each benchmark was run on each system and each microarchitecture 30 times, and measured with `perf stat`, measuring the events `instructions:u`, `cycles:u`, `branch-misses:u`, `L1-dcache-load-misses:u`, and `L1-icache-load-misses:u`, where available (but we only show results based on `cycles` and `instructions` here). The median of these runs is shown.

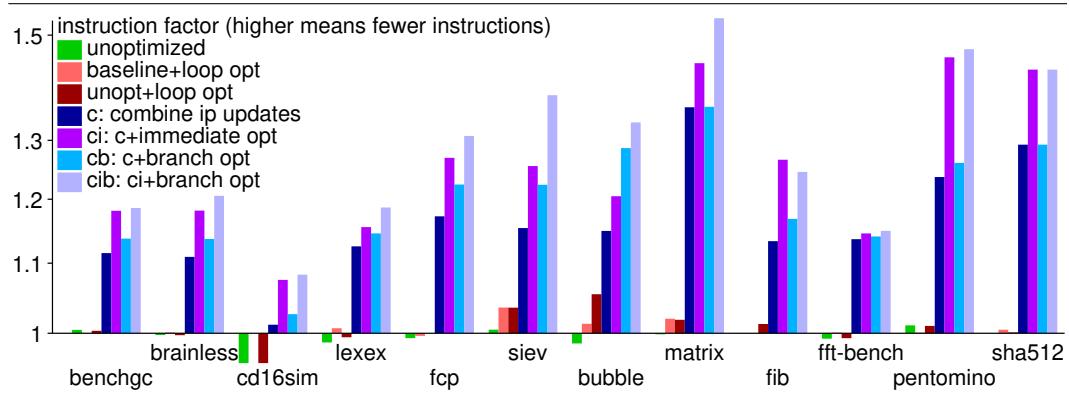


Figure 8 Reduction factor in the number of dynamically executed AMD64 instructions of various optimizations over *baseline*.

6 Results and discussion

6.1 Executed instructions

Figure 8 shows the effect of the IP update optimizations on the number of executed instructions on AMD64. For ARM A64 and RISC-V the results look similar.

For *unoptimized* and both loop optimization variants, the differences in executed instructions from the baseline are small, as expected (so small that sometimes you don't see the bar).

For most benchmarks *c* (combining IP updates) reduces the executed instructions, and *ci* (also optimize VM instructions with immediate operands) further reduces them (because more IP updates can be eliminated); adding *b* often has little effect on the number of executed instructions: in the usual case a load is replaced by an add.

On AMD64 and RISC-V where the IP updates have separate instructions, we can use the reduction in instructions to get an idea of the number of payload instructions in these benchmarks: If the unoptimized case has 1 IP update for n payload instructions, and the optimizations eliminate the proportion α of the IP updates on average, and the instruction reduction factor is f , we can compute $n = (1 - (1 - \alpha)f)/(f - 1)$. This leaves us with the problem of knowing α . However, if we assume that $\alpha = 1$, we get an upper bound for n ; e.g., for $f = 1.53$ (matrix), $n \leq 1.87$, while for $f = 1.2$ (brainless), $n \leq 5$. For matrix and siev *cib* eliminates all IP updates in the inner loop, and nearly all of the executed VM instructions of these benchmarks are in the inner loops, so α is close to 1, and n is close to 1.87 for matrix and close to 2.62 for siev.

6.2 Speedups from IP-update optimization variants

Figure 9 shows the speedups of the optimizations on Tiger Lake. As we will see, this is the microarchitecture where we typically see the best results, but Zen3 and Gracemont are not far off (Fig. 11).

On Tiger Lake, moving the IP updates to the start of each VM instruction (*unoptimized*) hurts a little on most benchmarks, but occasionally also helps.

Applying the loop optimization provides a speedup by a factor of about 2 on the three benchmarks (siev, bubble, matrix) that spend most of their time in short-to-medium length counted loops. However, for the huge loop body of sha512, the IP updates result in a

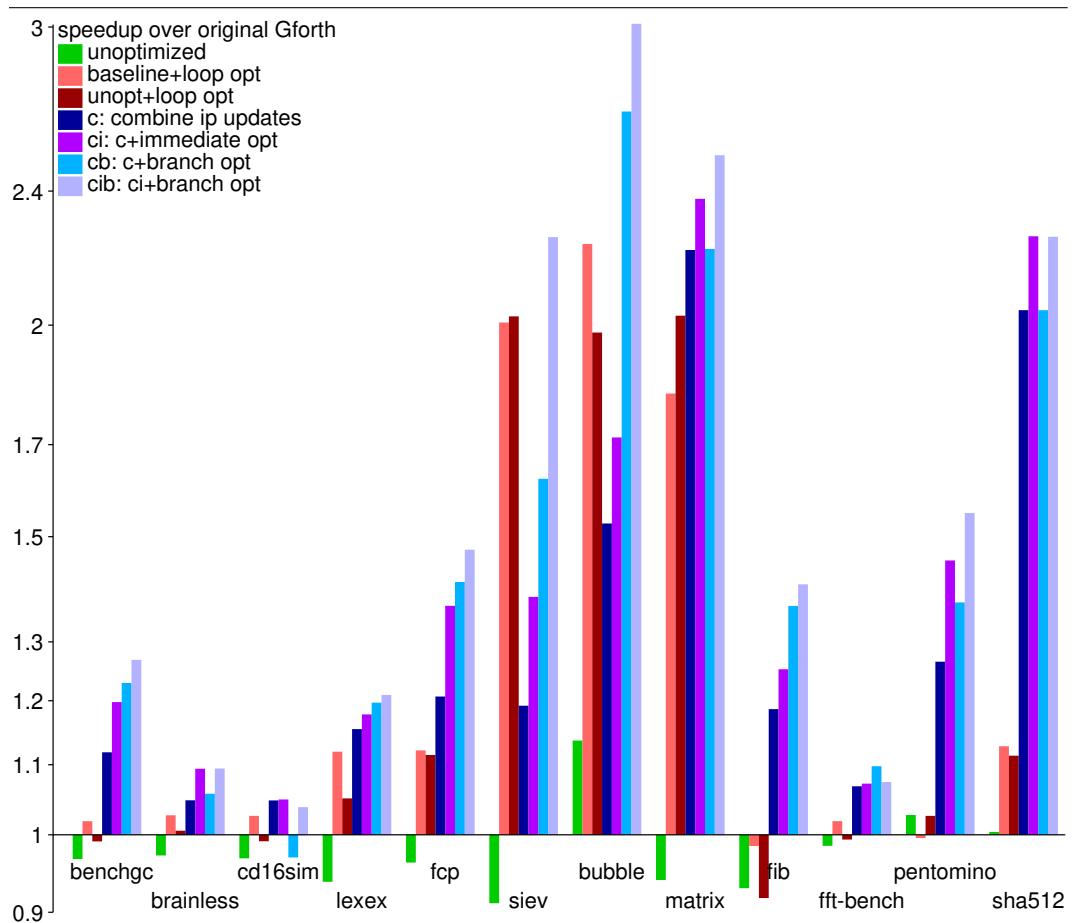


Figure 9 Speedup (reduction factor of execution cycles) on Tiger Lake of several optimizations over *baseline* (higher is better).

dependence chain that fills the processor buffers long before the loop-back branch breaks it, and the speedups of the loop optimization tend to be small. For fft-bench the inner loop is also a counted loop, but the loop body contains calls where the return breaks the IP dependence chain, so fft-bench does not benefit from the loop optimization. Pentomino hardly uses counted loops, so it cannot benefit from the loop optimization (as we implemented it). Most application benchmarks don't benefit, either.

Among the other variants, let us first look at *cib*: It dominates the loop optimization (whereas *c*, *ci* and *cb* don't, as demonstrated by *siev*). The speedups of *cib* depend on the benchmark, with *siev*, *bubble*, *matrix*, and *sha512* showing a speedup of > 2 on Tiger Lake, while the speedups on *fft-bench* and the application benchmarks are much more modest; in code where returns break the dependence chains, the main benefit of *cib* is the reduction in the number of executed instructions.

The results of *c*, *ci*, and *cb* are helpful in understanding the *cib* results: In short loops (*siev*, *bubble*) or code with many taken branches (*bubble*, *fib*) *cb* helps more than *ci*, because the loads of the branches are a large part of the latency chain in case of *ci*. By contrast, for programs with long loop bodies like *sha512*, the branch optimization does not work (it

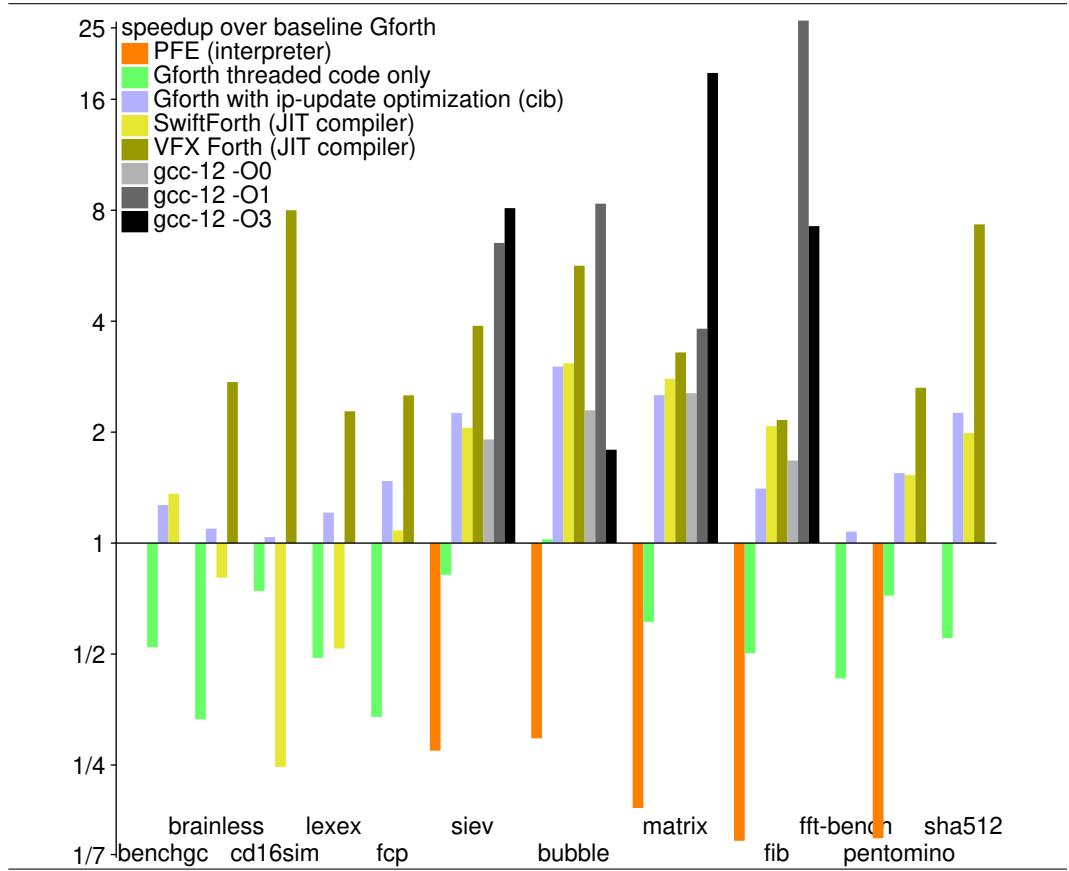


Figure 10 Speedup of several Forth systems and gcc over the Gforth baseline (higher is better), on Tiger Lake. If a benchmark does not work on a system, no bar is shown for the combination.

only covers short-distance branches) and we therefore see no difference between *ci* and *cib*. Pentomino has many branches, but many of them are long-distance branches as far as the branch optimization is concerned, so the benefit of the branch optimization is relatively small for this benchmark, and the benefit of the immediate optimization is more pronounced. Overall, for some benchmarks *ci* is better than *cb*, for others *cb* is better than *ci*; with the exception of cd16sim, both dominate *c*, and are dominated by *cib*. The difference is pretty big in some cases, so *cib* can be worth the additional implementation effort.

6.3 Comparison with other systems

Figure 10 compares a selection of the Gforth variants to several other Forth systems and to gcc. As in the other graphs, the baseline is Gforth version we started from.

Gforth with *cib* tends to be competitive with SwiftForth and with gcc -O0. SwiftForth shows some slowdowns for the application benchmarks despite executing significantly fewer instructions than Gforth; for cd16sim we identified the architectural pitfalls that it runs into⁷

⁷ I-cache/D-cache ping-pong from having instructions close to data, and `ret` mispredictions from using the return address of `call` as data instead of returning to it.

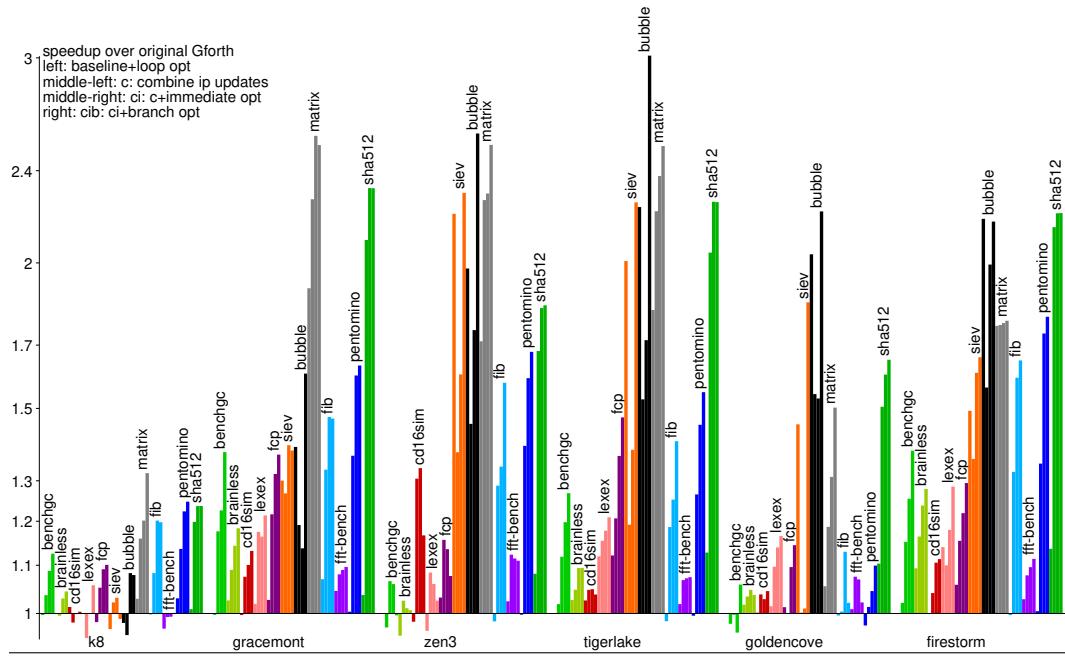


Figure 11 Speedup (reduction factor of execution cycles) of several optimizations over *baseline*; for each benchmark (colour), four bars show, from left to right: *l*, *c*, *ci*, *cib*.

and what implementation technique causes that, and reported it to the vendor. We did not investigate the SwiftForth performance on other benchmarks. The more sophisticated VFX Forth outperforms Gforth with *cib* usually by a factor of 2. Inlining of Forth definitions (performed by VFX, but not by Gforth) is particularly effective for *cd16sim*, leading to a speedup of VFX over *cib* by a factor of 8. Gforth with *cib* is a factor > 8 faster than PFE on the benchmarks where PFE works.

Gcc -O1 shows a factor 3–20 speedup over Gforth with *cib*, while for gcc -O3 the speedups over *cib* range from 0.6–8. For *bubble* the bad performance of gcc -O3 is caused by auto-vectorization, which exercises a slow hardware path for store-to-load-forwarding (due to partially overlapping accesses). We also looked at the gcc -O3 code for *fib*, but did not find an explanation for the slowdown compared to gcc -O1.

6.4 Speedups on different microarchitectures

Figure 11 shows a selection of the Gforth variants on several different microarchitectures. Most of them show similar speedups to Tiger Lake, which we discussed earlier.

One exception is Golden Cove (the P-Core of recent Intel CPUs); Golden Cove implements a hardware optimization that reduces the latency of adding a constant to zero cycles.⁸ This hardware optimization subsumes the *c* and *i* optimizations to some extent, and consequently,

⁸ <https://www.complang.tuwien.ac.at/anton/additions/>
<https://chipsandcheese.com/2021/12/21/gracemont-revenge-of-the-atom-cores/>

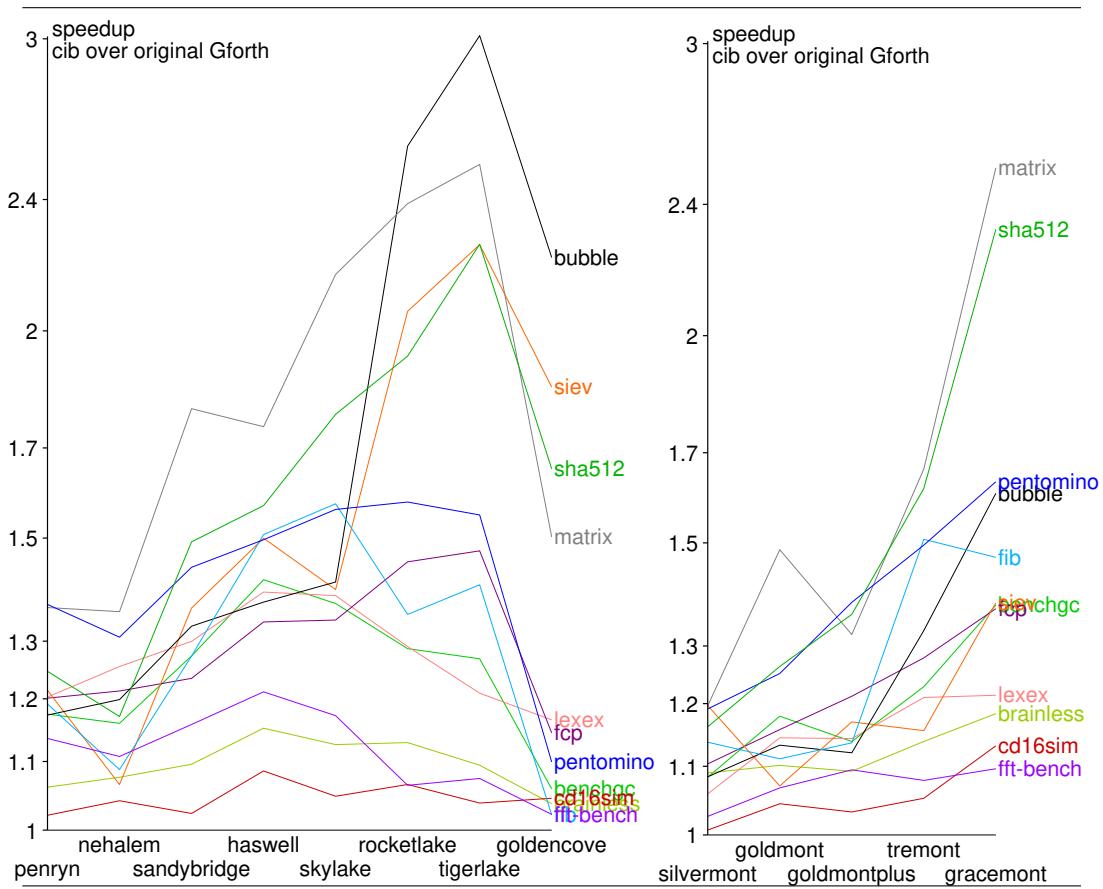


Figure 12 Speedups of *cib* over the baseline for different benchmarks on successive generations of Intel’s P-cores (left) and Intel’s E-cores (right).

we see lower speedups on Golden Cove than on Tiger Lake from these optimizations on a number of benchmarks. The benefit of the loop and branch optimization is still present, and shows up especially in code with short loops, such as *siev* and *bubble*, but the overall tendency is lower speedups from IP-update optimizations on Golden Cove. However, it still can pay off to apply IP-update optimizations, because CPUs with Golden Cove cores usually also have Gracemont E-cores, which benefit more from IP-update optimizations.

The other exception is the K8 microarchitecture (first released 2003). On the K8 the loop optimization tends to provide no benefit, and the other optimizations tend to provide benefits smaller than the reduction in instructions. This indicates that on the K8 the IP updates are not the critical path in instruction execution on any of these benchmarks.

We also looked at a variety of other CPUs (Fig. 12), and the tendency is that within a family of microarchitectures (e.g., Intel’s P-cores, with the exception of Golden Cove, as discussed above), the speedups from *cib* tend to be higher for more recent microarchitectures and lower for older microarchitectures. Along with the K8 results this explains why investigations on IP update optimizations have not been published earlier.

7 Applicability to other languages

In principle the IP-update optimizations can be applied to any VM implementation. In practice the benefit depends on how light-weight or heavy-weight the payload of your VM instructions is, on the characteristics of the executed programs.

Concerning program characteristics, loop-dominated programs benefit much more from the IP-update optimizations than call-dominated programs (see Section 6.2); but note that if you implement inlining or tail-call optimization, this can change call-dominated programs into loop-dominated programs.

Concerning the weight of VM instructions, IP update optimizations benefit VMs with lightweight instructions, such as Gforth, the OCaml interpreter, the JVM or WebAssembly. The lighter the payload is, the more these optimizations pay off.

By contrast, for a language implementation like Tcl with its heavy VM instructions, already dynamic superinstructions did not pay off; the speedup from the reduced dispatch overhead was small, and was compensated by increased I-cache misses [22].

Even if the VM instructions are middle-weight, we expect the benefit of the IP-update optimization to be small. E.g., if a VM instruction has an average payload of 10 instructions per VM instruction, the bottleneck will be in the payload (in the resource requirements, or in the latency), and the only benefit of the IP-update optimization will be to reduce the resource load, and that contribution will be relatively small (10%).

If you design a virtual machine that is lightweight enough that IP updates could be a bottleneck one day, it's a good idea to make it flexible enough make the loop optimization (Section 4.1) possible, which can be applied with relatively low effort.

8 Source code

The source code is in the git repository of Gforth:

```
git clone https://git.savannah.gnu.org/git/gforth.git
```

After that you can get the versions used for generating the data with:

```
cd gforth
# one of:
git checkout ecoop24-ip-updates-baseline #baseline
git checkout ecoop24-loopopt          #baseline+loop opt
git checkout ecoop24-ip-updates      #unopt, unopt+loop opt, c, ci, cb, cib
git checkout master #Gforth mainline
```

The main line of Gforth now uses *cib* by default.

You can find a package containing the checked out source code, binaries for AMD64, ARM A64, and RV64GC, benchmarks, and the resulting data on
<https://www.complang.tuwien.ac.at/anton/ip-updates.tar.xz>.

9 Related work

One difference between the approaches of Piumarta and Riccardi [17] and Ertl and Gregg [4] on selective inlining/dynamic superinstructions is that Piumarta and Riccardi eliminated the VM instruction slots of the VM instruction slots that are no longer needed for threaded-code dispatch. This eliminates as many IP updates as the *c* optimization, but Ertl and Gregg

expected that this “does not have much effect”. And indeed, the K8 results (similar to the hardware they used at the time) show a speedup ≤ 1.1 for c on most benchmarks (Section 6). However, on newer cores c provides speedups > 1.3 on some benchmarks, and we expect the same speedups from Piumarta and Riccardi’s instruction slot elimination. In any case, neither paper gives any performance evaluation of this issue, while the present work does and also explores additional optimizations: for loops, immediates, and branches.

kForth implements counted loops in the same way as the l optimization.⁹

There has been a significant body of work on combining VM instructions at VM-interpreter build time into (static) superinstructions [13, 18, 16, 11, 8, 3], which reduces instruction pointer updates, among other benefits. But again, none of these works have evaluated how much of the benefit is due to reducing IP updates.

More recent work on interpreter performance includes Rohou et al.’s reevaluation of the performance impact of indirect branches in the light of improvement in hardware indirect branch predictors [19], and Titzer’s work on an in-place interpreter for WebAssembly (which has been designed for translation) [21].

Larose et al. [14] argue that a sophisticated metacompiler (like RPython and Truffle) can optimize an AST interpreter written in a high-level language just as well as a VM interpreter. However, unless they completely eliminate all references to the AST or the VM code, they still have to maintain a pointer to the AST or VM code, and optimizing the IP updates is relevant.

In more ambitious earlier work [6], Ertl and Gregg eliminated the VM instruction pointer completely by eliminating all accesses to the threaded code: like the present work, it concatenates code snippets produced with gcc, but it patches constants and branch targets into the copied code snippets, making all access to the threaded code unnecessary. They found a median speedup by a factor 1.32 on a K7 (a 32-bit-only predecessor of the K8), quite an interesting contrast to the more modest speedups of the present IP-update optimization on the K8. However, this approach requires architecture-specific support for patching the constant and branch targets, whereas the present work is just as architecture-independent as dynamic superinstructions. This approach cannot fall back to threaded code, and therefore did not make it from proof-of-concept into a production feature of Gforth.

Xu and Kjolstad [24] have also used code snippets produced by a compiler and combined them, patching in constants and branch targets. The result also does not need a VM instruction pointer and its updates, and moreover, it uses the architecture’s call and return instructions (instead of jumps and indirect jumps). The price paid for this, like in Ertl and Gregg’s work [6], is architecture-specific code for patching the results.

The Maxine virtual machine [23] contains T1X, a compiler that uses snippets coming from a Java compiler, again without referencing the VM code, but also requiring architecture-specific code.

10 Conclusion

The IP-update optimization combination cib reduces the number of executed instructions by roughly a factor 1.2 on AMD64, ARM A64, and RISC-V. The effect on performance varies a lot across microarchitectures and benchmarks, between slowdowns by a factor 1.1 and speedups by a factor 3.01.

The reason for the more spectacular speedups is that, without cib , IP-update dependence chains become the critical path of execution on loop-dominated programs.

⁹ news:<us68iq\$3jsgk\$1@dont-email.me>

Another way to address this problem is the *loop optimization*: perform a loop-back branch to a location stored at loop entry, breaking the dependence chain. While the speedups from this optimization are not quite as spectacular as those from *cib*, and this optimization speeds up only some benchmarks, it is much easier to implement.

References

- 1 James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.
- 2 Helmut Eller. Optimizing interpreters with superinstructions. Diplomarbeit, TU Wien, 2005. URL: <https://www.complang.tuwien.ac.at/Diplomarbeiten/eller05.ps.gz>.
- 3 M. Anton Ertl and David Gregg. Implementation issues for superinstructions in Gforth. In *EuroForth 2003 Conference Proceedings*, 2003. URL: <https://www.complang.tuwien.ac.at/papers/ertl%26gregg03euroforth.ps.gz>.
- 4 M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, 2003. URL: <https://www.complang.tuwien.ac.at/papers/ertl%26gregg03.ps.gz>.
- 5 M. Anton Ertl and David Gregg. Combining stack caching with dynamic superinstructions. In *Interpreters, Virtual Machines and Emulators (IVME '04)*, pages 7–14, 2004. URL: <https://www.complang.tuwien.ac.at/papers/ertl%26gregg04ivme.ps.gz>.
- 6 M. Anton Ertl and David Gregg. Retargeting JIT compilers by using C-compiler generated executable code. In *Parallel Architecture and Compilation Techniques (PACT' 04)*, pages 41–50, 2004. URL: <https://www.complang.tuwien.ac.at/papers/ertl%26gregg04pact.ps.gz>.
- 7 M. Anton Ertl and David Gregg. Stack caching in Forth. In *21st EuroForth Conference*, pages 6–15, 2005. URL: <https://www.complang.tuwien.ac.at/papers/ertl%26gregg05.ps.gz>.
- 8 M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. *vmgen* — a generator of efficient virtual machine interpreters. *Software—Practice and Experience*, 32(3):265–294, 2002. URL: <https://www.complang.tuwien.ac.at/papers/ertl+02.ps.gz>.
- 9 M. Anton Ertl and Bernd Paysan. Gforth. Software, version 0.7.9_20240821., swId: swh:1:dir:61eb3b71325060fe2e01f5e819eb0bec959e5bf0 (visited on 2024-09-02). URL: <https://git.savannah.gnu.org/cgit/gforth.git>.
- 10 M. Anton Ertl and Bernd Paysan. ip-updates. Collection, version 7. (visited on 2024-09-02). URL: <https://www.complang.tuwien.ac.at/anton/ip-updates.tar.xz>.
- 11 David Gregg and John Waldron. Primitive sequences in general purpose Forth programs. In *18th EuroForth Conference*, pages 24–32, 2002. URL: <http://www.complang.tuwien.ac.at/anton/euroforth2002/papers/gregg.ps.gz>.
- 12 Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 185–200, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3062341.3062363.
- 13 R. J. M. Hughes. Super-combinators. In *Conference Record of the 1980 LISP Conference, Stanford, CA*, pages 1–11, New York, 1982. ACM.
- 14 Octave Larose, Sophie Kaleba, Humphrey Burchell, and Stefan Marr. AST vs. bytecode: Interpreters in the age of meta-compilation. *Proc. ACM Program. Lang.*, 7(OOPSLA2), October 2023. doi:10.1145/3622808.
- 15 Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, first edition, 1997.
- 16 Henrik Nässén, Mats Carlsson, and Konstantinos Sagonas. Instruction merging and specialization in the SICStus Prolog virtual machine. In *Principles and Practice of Declarative Programming (PPDP01)*, 2001. URL: <http://www.cs.uu.se/%7Ekostis/Papers/sicstus.ps.gz>.

14:26 The Performance Effects of Virtual-Machine Instruction Pointer Updates

- 17 Ian Piumarta and Fabio Riccardi. Optimizing direct threaded code by selective inlining. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 291–300, 1998. URL: ftp://ftp.inria.fr/INRIA/Projects/SOR/papers/1998/ODCSI_pldi98.ps.gz.
- 18 Todd A. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Principles of Programming Languages (POPL '95)*, pages 322–332, 1995.
- 19 Erven Rohou, Bharath Narasimha Swamy, and André Seznec. Branch prediction and the performance of interpreters — don't trust folklore. In *Code Generation and Optimization (CGO)*, 2015. URL: <https://hal.inria.fr/hal-01100647/document>.
- 20 Markku Rossi and Kengatharan Sivalingam. A survey of instruction dispatch techniques for byte-code interpreters. Technical Report TKO-C79, Faculty of Information Technology, Helsinki University of Technology, May 1996. URL: <http://www.cs.hut.fi/~cessu/papers/dispatch.ps>.
- 21 Ben L. Titzer. A fast in-place interpreter for WebAssembly. *Proc. ACM Program. Lang.*, 6(OOPSLA2):148:1–148:27, 2022.
- 22 Benjamin Vitale and Tarek S. Abdelrahman. Catenation and specialization for Tcl virtual machine performance. In *IVME '04 Proceedings*, pages 42–50, 2004.
- 23 Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. Maxine: An approachable virtual machine for, and in, Java. *ACM Transactions on Architecture and Code Optimization*, 9(4):30:1–30:24, January 2013.
- 24 Haoran Xu and Fredrik Kjolstad. Copy-and-patch compilation. *Proc. ACM Program. Lang.*, 5(OOPSLA):136:1–136:30, October 2021. URL: <https://fredrikbk.com/publications/copy-and-patch.pdf>.

Rose: Composable Autodiff for the Interactive Web

Sam Estep  

Software and Societal Systems Department, Carnegie Mellon University, Pittsburgh, PA, USA

Wode Ni  

Software and Societal Systems Department, Carnegie Mellon University, Pittsburgh, PA, USA

Raven Rothkopf  

Barnard College, Columbia University, New York, NY, USA

Joshua Sunshine  

Software and Societal Systems Department, Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

Reverse-mode automatic differentiation (autodiff) has been popularized by deep learning, but its ability to compute gradients is also valuable for interactive use cases such as bidirectional computer-aided design, embedded physics simulations, visualizing causal inference, and more. Unfortunately, the web is ill-served by existing autodiff frameworks, which use autodiff strategies that perform poorly on dynamic scalar programs, and pull in heavy dependencies that would result in unacceptable webpage sizes. This work introduces Rose, a lightweight autodiff framework for the web using a new hybrid approach to reverse-mode autodiff, blending conventional tracing and transformation techniques in a way that uses the host language for metaprogramming while also allowing the programmer to explicitly define reusable functions that comprise a larger differentiable computation. We demonstrate the value of the Rose design by porting two differentiable physics simulations, and evaluate its performance on an optimization-based diagramming application, showing Rose outperforming the state-of-the-art in web-based autodiff by multiple orders of magnitude.

2012 ACM Subject Classification Software and its engineering → Compilers; Information systems → Web applications; Software and its engineering → Domain specific languages; Computing methodologies → Symbolic and algebraic manipulation; Software and its engineering → Formal language definitions; General and reference → Performance; Computing methodologies → Neural networks; General and reference → General conference proceedings

Keywords and phrases Automatic differentiation, differentiable programming, compilers, web

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.15

Related Version *Full Version:* <https://arxiv.org/abs/2402.17743> [10]

Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact):*

<https://doi.org/10.4230/DARTS.10.2.7>

Software (Source Code): <https://github.com/rose-lang/rose>

archived at [swh:1:dir:bc091e3b381dd680e389e14729302848edd7d0aa](https://scholar.archive.org/2024/02/14/14729302848edd7d0aa)

Funding This material is based upon work supported by the Aqueduct Foundation and by National Science Foundation under Grant Numbers 1910264, 2119007, and 2150217.

Acknowledgements Thanks to Adam Paszke for corresponding about JAX and Dex. The Rose icons were created by Aaron Weiss; we use them via the CC BY 4.0 license.

1 Introduction

The web provides a platform for interactive experiences with a uniquely low barrier to usage, because the browser obviates the need for software installation by automatically downloading JavaScript code and running it securely on the client. Industry tools like Google Slides [13] and Figma [11], as well as experimental tools like Sketch-n-Sketch [17]

 © Sam Estep, Wode Ni, Raven Rothkopf, and Joshua Sunshine;
licensed under Creative Commons License CC-BY 4.0
38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 15; pp. 15:1–15:27



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

and Penrose [54], leverage this platform to enable authoring of visual media. Interactive explainers like Red Blob Games [34], Bartosz Ciechanowski’s work [9], and Bret Victor’s “Explorable Explanations” [50] use the web to help people understand complicated ideas in depth, building up a causal mental model by using sliders to manipulate values and immediately see the effects.

Many of these interactions are fairly simple: often the user just drags a slider back and forth, manipulating a parameter in, for instance, a small physical simulation. But there is room for much richer interactions. An early exploration was g9.js [52], which lets the user directly drag around visual shapes, and automatically propagates those changes backward to modify the underlying parameters driving the visualization. This idea of *bidirectional editing* or *bidirectional transformations* [3] is quite powerful. Some more recent work [8] has explored bidirectional editing in computer-aided design (CAD) via *automatic differentiation (autodiff)*, a technique for efficiently computing derivatives of numerical functions. Autodiff has become popularized over the past few years by machine learning (ML) frameworks such as TensorFlow [2], PyTorch [32], and JAX [12].

Autodiff engines built for ML are focused on high throughput for functions composed of a relatively small number of operations on relatively large tensors. They use *reverse-mode* autodiff to compute the gradient of a *loss function* in an iterative loop, using a numerical optimization algorithm like stochastic gradient descent or Adam [25] to update the parameters of the ML model until the loss value is sufficiently reduced. The loss function is usually chosen to be parallelizable on a GPU. These characteristics do not generally apply to other domains, which often involve *scalar programs* [22] on which overhead between operations would dominate any tensor-level attempts at parallelism.

For scalar programs, program transformation tools are far more appropriate; examples include Tapenade [16] for Fortran and C, Zygote [21] for Julia [7], and Enzyme [29] for LLVM [26]. These tools consume and emit code that deals directly with scalars, reducing expressiveness limitations and operation-level overhead at the expense of the parallelism that ML frameworks gain by specializing to tensor operations. They typically leverage heavy modern compiler technology, using various optimization passes on the program after (and sometimes before) differentiation.

But in the interactive web setting, none of these existing points in the design space are appropriate. We are operating in an environment that is

- **dynamic:** the goal is to let the user author content or build up their mental causal model, by (either implicitly through direct manipulation or explicitly through writing code) specifying a differentiable function themselves. The autodiff engine must operate *online*, differentiating functions directly inside of the user’s browser.
- **bandwidth-constrained:** because of the no-install model described above, any JavaScript or WebAssembly code used for autodiff must be shipped over the network to the user’s browser. Heavyweight components are unacceptable because their bandwidth requirements would exacerbate page load times beyond the user’s patience.
- **latency-constrained:** the system must respond to the user’s manipulation of the differentiable function definition, at interactive speed. What we care about is not just the performance of the synthesized gradient, but the sum of that latency with the latency to synthesize the gradient in the first place; *quantitative* differences like a slow “compilation” step result in *qualitative* differences in the kinds of interaction possible.

All existing autodiff tools, including web-focused tools like TensorFlow.js [46], fall short on at least one of these constraints: they impose large constant factors for scalar programs, or depend on giant codebases that are difficult to package for the web and result in large bundles, or are too slow to use in an interactive setting, or some combination of these.

To address this gap, we present **Rose**,¹ a scalar-focused autodiff engine for the web that achieves fast compilation time and high generated code performance in a small bundle. As we will describe in Section 4, Rose is a hybrid autodiff system [22] which blends together techniques from tracing and program transformation before emitting WebAssembly [15]. Unlike prior program transformation approaches that take advantage of heavyweight compiler optimization toolchains, we produce efficient Rose IR before differentiating by using JavaScript as a *metaprogramming environment*, somewhat similar to tracing in popular ML frameworks. But unlike prior tracing approaches that expand all operations into one large graph, we reduce generated code size and thus compilation time by allowing the user to explicitly define *composable functions* that can be nested and reused. Our primary contributions are as follows:

- We establish the importance of, and constraints imposed by, the interactive web setting, and articulate how those constraints translate to system requirements for autodiff in such a setting.
- We describe a novel system design that satisfies these requirements using a careful blend of tracing with program transformation.
- We present experiments demonstrating how each component of our design is key to achieving the requirements we have laid out.
- We publish Rose, an open-source software package implementing this design for others to consume and build upon.

The rest of this paper is structured as follows. In Section 2 we give relevant general mathematical background information about autodiff; we introduce a running example that we then implement in Section 3, which discusses Rose from a user perspective. Then Section 4 discusses the novel design of Rose, focusing on the high level because that is our more interesting contribution, but also describing some low-level details of autodiff for the curious reader. Section 5 describes the experiments we conducted with results showing why this design is key to achieving our design goals. Finally we discuss related work in Section 6, and conclude with future work in Section 7.

2 Background

To illustrate the basic ideas of reverse-mode autodiff, we'll walk through the classic example of using gradients to perform linear regression via least-squares optimization. Nothing in this section is new. Almost all the content here can be found in standard textbooks for calculus, linear algebra, and convex optimization; all the rest can be found in the research literature on autodiff [39].

Suppose we have n measurements $\mathbf{y} \in \mathbb{R}^n$ of a dependent variable, each corresponding to one of n data points $\mathbf{x}_1, \dots, \mathbf{x}_n \in \mathbb{R}^m$. These data can be assembled into a matrix $\mathbf{X} \in \mathbb{R}^{n \times (m+1)}$ defined by

$$\mathbf{X} = \begin{bmatrix} 1 & \mathbf{x}_1^\top \\ \vdots & \vdots \\ 1 & \mathbf{x}_n^\top \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & \cdots & x_{1m} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \cdots & x_{nm} \end{bmatrix}.$$

We would like to predict the dependent variable as a linear function $\hat{\mathbf{y}} = \mathbf{X}\beta$ where the parameters $\beta \in \mathbb{R}^{m+1}$ are chosen to minimize the sum of squares of the errors $\varepsilon = \mathbf{y} - \hat{\mathbf{y}}$.

¹ Not to be confused with the ROSE (all caps) compiler infrastructure. [38]

That is, one would like to find an optimal solution to the optimization problem

$$\min_{\beta \in \mathbb{R}^{m+1}} f(\beta) \quad \text{where} \quad f(\beta) = \|\varepsilon\|^2 = \|\mathbf{y} - \hat{\mathbf{y}}\|^2 = \|\mathbf{y} - \mathbf{X}\beta\|^2.$$

Applying convex optimization theory here is standard so we won't belabor it, but because this particular f is differentiable, convex, and smooth, there exists a step size $\eta > 0$ such that if we start with any $\beta_0 \in \mathbb{R}^{m+1}$ and iteratively compute $\beta_{i+1} = \beta_i - \eta \nabla f(\beta_i)$, then

$$f(\beta_{i+1}) \leq f(\beta_i) \quad \forall i \in \mathbb{N}, \quad \text{and} \quad \lim_{i \rightarrow \infty} f(\beta_i) \leq f(\beta) \quad \forall \beta \in \mathbb{R}^{m+1}.$$

This is gradient descent. Crucially, it depends on being able to compute the gradient ∇f .

2.1 The vector-Jacobian product

As briefly mentioned in Section 1, reverse-mode autodiff is a general method for computing gradients, which takes in an algorithm to compute a function, and returns an algorithm to compute its gradient. Unlike other approaches to compute derivatives, the power of autodiff lies in its *compositionality* and *efficiency*: we naturally express functions by composing together smaller functions. If we possess an algorithm that computes a given function with a given time complexity, reverse-mode autodiff gives us an algorithm to compute its derivative with the same time complexity, in a way that can be directly composed with derivatives for other functions. For example, in least-squares we compose together three functions

$$\begin{array}{lll} \xi : \mathbb{R}^{m+1} \rightarrow \mathbb{R}^n & \varphi : \mathbb{R}^n \rightarrow \mathbb{R}^n & \psi : \mathbb{R}^n \rightarrow \mathbb{R} \\ \xi(\beta) = \mathbf{X}\beta & \varphi(\hat{\mathbf{y}}) = \mathbf{y} - \hat{\mathbf{y}} & \psi(\varepsilon) = \|\varepsilon\|^2 \end{array}$$

to form $f = \psi \circ \varphi \circ \xi$. Clearly, the gradient itself is insufficient to express ∇f compositionally, because ξ and φ are not scalar-valued and thus do not have gradients. So we must first have a compositional definition for the derivative.

The *Jacobian* of a function $\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^m$ is the matrix-valued function $\mathbf{J}_\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^{m \times n}$ defined by

$$\mathbf{J}_\mathbf{f}(\mathbf{x}) = \begin{bmatrix} \frac{\partial \mathbf{f}}{\partial x_1} & \dots & \frac{\partial \mathbf{f}}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \dots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}.$$

From this, if we fix $\mathbf{x} \in \mathbb{R}^n$ then we can define a function $\text{vjp}_\mathbf{f}^\mathbf{x} : \mathbb{R}^{1 \times m} \rightarrow \mathbb{R}^{1 \times n}$, called the *vector-Jacobian product (VJP)*, operating on row vectors called *adjoints* by $\text{vjp}_\mathbf{f}^\mathbf{x}(\hat{\mathbf{y}}) = \hat{\mathbf{y}} \mathbf{J}_\mathbf{f}(\mathbf{x})$. In the special case of $m = 1$ we can recover the gradient by $\nabla \mathbf{f}(\mathbf{x}) = \text{vjp}_\mathbf{f}^\mathbf{x}(1)^\top$, but unlike the gradient, this notion of a derivative actually composes. For instance, if we also have $\mathbf{g} : \mathbb{R}^m \rightarrow \mathbb{R}^p$, then

$$\text{vjp}_{\mathbf{g} \circ \mathbf{f}}^\mathbf{x} = \text{vjp}_\mathbf{f}^\mathbf{x} \circ \text{vjp}_\mathbf{g}^\mathbf{y} \quad \text{where} \quad \mathbf{y} = \mathbf{f}(\mathbf{x}).$$

This is the chain rule for reverse-mode autodiff, so-called because it composes $\text{vjp}_\mathbf{f}$ and $\text{vjp}_\mathbf{g}$ in the reverse order of how \mathbf{f} and \mathbf{g} themselves were originally composed. As shown here, computing the derivative $\text{vjp}_{\mathbf{g} \circ \mathbf{f}}$ depends on computing the original function \mathbf{f} itself, so in practice the term "VJP" is sometimes actually used to refer to the mapping

$$\begin{aligned} \mathbb{R}^n &\rightarrow \mathbb{R}^m \times (\mathbb{R}^{1 \times m} \rightarrow \mathbb{R}^{1 \times n}) \\ \mathbf{x} &\mapsto (\mathbf{f}(\mathbf{x}), \text{vjp}_\mathbf{f}^\mathbf{x}) \end{aligned}$$

that returns both the output of the original function – which we call a *primal* value to contrast it with the VJP's adjoints – and the VJP function.

We'll provide a more general set of composable VJPs in Sections 4.1 and 4.3, but for this example, we can derive the VJPs

$$\begin{array}{lll} \text{vjp}_\xi^\beta : \mathbb{R}^{1 \times n} \rightarrow \mathbb{R}^{1 \times (m+1)} & \text{vjp}_\varphi^{\hat{y}} : \mathbb{R}^{1 \times n} \rightarrow \mathbb{R}^{1 \times n} & \text{vjp}_\psi^\epsilon : \mathbb{R} \rightarrow \mathbb{R}^{1 \times n} \\ \text{vjp}_\xi^\beta(\ddot{\mathbf{y}}) = \ddot{\mathbf{y}} \mathbf{X} & \text{vjp}_\varphi^{\hat{y}}(\ddot{\epsilon}) = -\ddot{\epsilon} & \text{vjp}_\psi^\epsilon(\ddot{\sigma}) = 2\ddot{\sigma}\epsilon^\top \end{array}$$

of the functions we decomposed earlier. One key property to notice here is that, given algorithms to compute ξ , φ , and ψ , we immediately have algorithms to compute vjp_ξ , vjp_φ , and vjp_ψ , respectively, with the same time complexities. For instance, ξ is $O(mn)$ with naïve matrix multiplication, as is vjp_ξ . This is not too surprising, since that is also the same time complexity as directly computing and multiplying by \mathbf{J}_ξ . But the time complexity for both φ and ψ is $O(n)$, as are the formulas given above for vjp_φ and vjp_ψ , in contrast to the $O(n^2)$ cost of naïvely computing \mathbf{J}_φ or \mathbf{J}_ψ . This is because those Jacobians are *sparse*; the ability of reverse-mode autodiff to preserve time complexity in the presence of sparse Jacobians is called the *cheap gradient principle*.

In any case, from these simpler VJPs we can easily compose the gradient of f as

$$\begin{aligned} \nabla f(\beta) &= \text{vjp}_f^\beta(1)^\top = \text{vjp}_\xi^\beta(\text{vjp}_\varphi^{\hat{y}}(\text{vjp}_\psi^\epsilon(1)))^\top = (-2\epsilon^\top \mathbf{X})^\top = 2\mathbf{X}^\top(\mathbf{X}\beta - \mathbf{y}) \\ \text{where } \hat{\mathbf{y}} &= \epsilon(\beta) = \mathbf{X}\beta \quad \text{and} \quad \epsilon = \varphi(\hat{\mathbf{y}}) = \mathbf{y} - \hat{\mathbf{y}}. \end{aligned}$$

2.2 The Jacobian-vector product

We've talked about the VJP used for reverse-mode autodiff, which is the more useful for optimization, but also the more challenging to implement and specify. Rose allows users to specify custom derivatives for reasons described in Section 3.2, so to reduce user burden, we allow those custom derivatives to be defined using the simpler *Jacobian-vector product (JVP)* instead of the VJP. In Section 4.1 we'll discuss the actual program transformation used to derive the VJP from the JVP [39], but here we first lay out the mathematical groundwork.

For $\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^m$, the JVP of \mathbf{f} at $\mathbf{x} \in \mathbb{R}^n$ is a function $\text{jvp}_{\mathbf{f}}^x : \mathbb{R}^n \rightarrow \mathbb{R}^m$ that operates on column vectors called *tangents* by $\text{jvp}_{\mathbf{f}}^x(\dot{\mathbf{x}}) = \mathbf{J}_{\mathbf{f}}(\mathbf{x})\dot{\mathbf{x}}$. In the special case of $n = 1$ we can recover the ordinary derivative by $\mathbf{f}'(\mathbf{x}) = \text{jvp}_{\mathbf{f}}^x(1)$. But more generally, given $\mathbf{g} : \mathbb{R}^m \rightarrow \mathbb{R}^p$ we also have a chain rule

$$\text{jvp}_{\mathbf{g} \circ \mathbf{f}}^x = \text{jvp}_{\mathbf{g}}^y \circ \text{jvp}_{\mathbf{f}}^x \quad \text{where } \mathbf{y} = \mathbf{f}(\mathbf{x})$$

that composes $\text{jvp}_{\mathbf{f}}$ and $\text{jvp}_{\mathbf{g}}$ in the same order as \mathbf{f} and \mathbf{g} , hence the name “forward-mode.” This is much simpler computationally, because while reverse-mode needed to compose together the VJPs themselves until the end when it could call them with the final adjoint value, forward-mode can simply use the mapping

$$\begin{aligned} \mathbb{R}^n \times \mathbb{R}^n &\rightarrow \mathbb{R}^m \times \mathbb{R}^m \\ (\mathbf{x}, \dot{\mathbf{x}}) &\mapsto (\mathbf{f}(\mathbf{x}), \text{jvp}_{\mathbf{f}}^x(\dot{\mathbf{x}})) \end{aligned}$$

to package together the primal and tangent values.

The judgment is somewhat subjective, but we invite the reader to decide for themselves whether the VJPs derived earlier or these JVPs

$$\begin{array}{lll} \text{jvp}_\xi^\beta : \mathbb{R}^{m+1} \rightarrow \mathbb{R}^n & \text{jvp}_\varphi^{\hat{y}} : \mathbb{R}^n \rightarrow \mathbb{R}^n & \text{jvp}_\psi^\epsilon : \mathbb{R}^n \rightarrow \mathbb{R} \\ \text{jvp}_\xi^\beta(\dot{\beta}) = \mathbf{X}\dot{\beta} & \text{jvp}_\varphi^{\hat{y}}(\dot{\hat{y}}) = -\dot{\hat{y}} & \text{jvp}_\psi^\epsilon(\dot{\epsilon}) = 2\epsilon^\top \dot{\epsilon} \end{array}$$

are closer to the original definitions of ξ , φ , and ψ .

When packaging together \mathbf{f} with $\text{jvp}_{\mathbf{f}}$, it is convenient to represent the pair $(\mathbf{x}, \dot{\mathbf{x}}) \in \mathbb{R}^n \times \mathbb{R}^n$ as the single vector $\bar{\mathbf{x}} = \mathbf{x} + \dot{\mathbf{x}}\varepsilon \in \mathbb{D}^n$, making use of the infinitesimal element ε in the *dual numbers* defined by the commutative algebra $\mathbb{D} = \{a + b\varepsilon \mid a, b \in \mathbb{R}\}$ where $\varepsilon^2 = 0$. For dual numbers $\bar{x} = x + \dot{x}\varepsilon \in \mathbb{D}$ and $\bar{y} = y + \dot{y}\varepsilon \in \mathbb{D}$, the arithmetic operations

$$\begin{aligned}\bar{x} + \bar{y} &= x + y + (\dot{x} + \dot{y})\varepsilon \\ \bar{x} - \bar{y} &= x - y + (\dot{x} - \dot{y})\varepsilon \\ \bar{x}\bar{y} &= xy + (\dot{x}y + x\dot{y})\varepsilon \\ \frac{\bar{x}}{\bar{y}} &= \frac{x}{y} + \frac{\dot{x}y - x\dot{y}}{y^2}\varepsilon \quad \text{where } y \neq 0\end{aligned}$$

correspond directly to the JVPs of the corresponding arithmetic operations on real numbers. This allows us to define what we'll call the *dual JVP*

$$\begin{aligned}\overline{\text{jvp}}_{\mathbf{f}} : \mathbb{D}^n &\rightarrow \mathbb{D}^m \\ \overline{\text{jvp}}_{\mathbf{f}}(\mathbf{x} + \dot{\mathbf{x}}\varepsilon) &= \mathbf{f}(\mathbf{x}) + \text{jvp}_{\mathbf{f}}^{\mathbf{x}}(\dot{\mathbf{x}})\varepsilon\end{aligned}$$

which operates on column vectors of dual numbers. In this framing, specifying the JVPs of our three functions

$$\begin{array}{lll}\overline{\text{jvp}}_{\xi} : \mathbb{D}^{m+1} &\rightarrow \mathbb{D}^n & \overline{\text{jvp}}_{\varphi} : \mathbb{D}^n &\rightarrow \mathbb{D}^n & \overline{\text{jvp}}_{\psi} : \mathbb{D}^n &\rightarrow \mathbb{D} \\ \overline{\text{jvp}}_{\xi}(\bar{\beta}) &= \mathbf{X}\bar{\beta} & \overline{\text{jvp}}_{\varphi}(\bar{\mathbf{y}}) &= \mathbf{y} - \bar{\mathbf{y}} & \overline{\text{jvp}}_{\psi}(\bar{\varepsilon}) &= \bar{\varepsilon}^\top \bar{\varepsilon}\end{array}$$

becomes almost trivial. So when we refer to the JVP in Rose, we're always talking about this dual JVP, not the raw JVP which would operate on real numbers.

2.3 The Hessian

While gradient descent is a first-order method that only makes use of the gradient, other optimization techniques such as Newton's method also need the *Hessian*, which turns $f : \mathbb{R}^n \rightarrow \mathbb{R}$ into a matrix-valued function $\mathbf{H}_f : \mathbb{R}^n \rightarrow \mathbb{R}^{n \times n}$ yielding all the second-order partial derivatives of f at a given point. The JVP and VJP can be used together to define the Hessian, which is actually just the Jacobian of the gradient; that is, $\mathbf{H}_f = \mathbf{J}_{\nabla f}$. We already know how to use the VJP of a function to compute its gradient. To compute the Jacobian, we just need to observe that the i^{th} row of the Jacobian is equal to the JVP applied to the i^{th} basis element \mathbf{e}_i of \mathbb{R}^n :

$$\mathbf{H}_f(\mathbf{x}) = [\text{jvp}_{\nabla f}^{\mathbf{x}}(\mathbf{e}_1) \quad \cdots \quad \text{jvp}_{\nabla f}^{\mathbf{x}}(\mathbf{e}_n)] \quad \text{where } \nabla f(\mathbf{x}) = \text{vjp}_{\mathbf{f}}^{\mathbf{x}}(1)^\top$$

3 Using Rose

Now that we've discussed the mathematical ideas behind Rose, in this section we'll describe how programmers use Rose to understand how it fits into the context described in Section 1. Then in Section 4 we'll describe our design that blends together tracing with program transformation to fit into this context.

Listing 1 shows a comprehensive end-to-end example using Rose to perform gradient descent to solve the linear regression problem laid out in Section 2. This entire example is JavaScript code, which makes use of Rose as a library; lines 1–2 use standard JavaScript `import` statements to pull in definitions from the Rose library. Lines 3–10 use arithmetic primitives from Rose to implement the loss function from Section 2:

$$f(\beta) = \|\mathbf{y} - \mathbf{X}\beta\|^2 = \sum_{i=1}^n \left(y_i - \beta_0 - \sum_{j=1}^m x_{ij}\beta_j \right)^2$$

```

1 import { Real, Vec, compile, fn, struct, vjp } from "rose";
2 import { add, mul, sub, sum } from "rose";
3 const sqr = (x) => mul(x, x);
4 const leastSquares = ({ m, n }) => fn([
5   x: Vec(n, Vec(m, Real)), y: Vec(n, Real),
6   b0: Real, b: Vec(m, Real),
7 ], Real, ({ x, y, b0, b }) => sum(n, (i) => {
8   const yHat = add(b0, sum(m, (j) => mul(x[i][j], b[j])));
9   return sqr(sub(y[i], yHat));
10 }));
11 const linearRegression = async ({ x, y, eta }) => {
12   const [n, m] = [y.length, x[0].length];
13   const Beta = struct({ b0: Real, b: Vec(m, Real) });
14   const f = leastSquares({ m, n });
15   const g = fn([Beta], Real, ({ b0, b }) => f({ x, y, b0, b }));
16   const h = fn([Beta], Beta, (beta) => vjp(g)(beta).grad(1));
17   const grad = await compile(h);
18   let b0 = 0; let b = Array(m).fill(0);
19   for (;;) {
20     const beta = grad({ b0, b });
21     const bb0 = b0 - eta * beta.b0;
22     const bb = b.map((bi, i) => bi - eta * beta.b[i]);
23     if (bb0 === b0 && bb.every((bi, i) => bi === b[i])) break;
24     b0 = bb0; b = bb;
25   }
26   return { b0, b };
27 };
28 console.log(await linearRegression({ eta: 1e-4,
29   x: [[10], [8], [13], [9], [11], [14], [6], [4], [12], [7], [5]],
30   y: [8.04, 6.95, 7.58, 8.81, 8.33, 9.96, 7.24, 4.26, 10.84, 4.82, 5.68],
31 }));

```

■ Listing 1 Using Rose to do linear regression on the first dataset in Anscombe’s quartet [4].

Lines 4–7 define the type of the `leastSquares` function, which takes as input $\mathbf{x}_1, \dots, \mathbf{x}_n \in \mathbb{R}^m$, as well as $\mathbf{y} \in \mathbb{R}^n$ and $\beta \in \mathbb{R}^{m+1}$, and returns a scalar. Lines 11–27 wrap around that low-level function to provide a high-level method to perform linear regression, which is then used by lines 28–31. More specifically, line 13 uses Rose to define the type of $\beta \in \mathbb{R}^{m+1}$, a vector with m elements plus an additional scalar bias. Line 14 uses the earlier `leastSquares` definition to get a Rose function for the specific $m, n \in \mathbb{N}$ needed, and line 15 partially applies that function using the provided `x` and `y` values as constants. Line 16 uses Rose’s builtin `vjp` function to take the gradient of that partially applied loss function. While mathematically it can be useful to distinguish column and row vectors, the Rose library does not, so the VJP directly produces the gradient. Line 17 compiles that gradient function to WebAssembly, producing a function that can be called with concrete standard JavaScript values instead of abstract Rose values. Finally, lines 18–26 perform gradient descent by calling the compiled `grad` function.

As demonstrated by the above examples, Rose works by letting the user define *differentiable functions* of the form `fn(paramTypes, returnType, body)`. We'll discuss this more in Section 4, but the high-level idea from a user perspective is that normal JavaScript functions like the one defined on line 3 of Listing 1 roughly correspond to what one might think of as *macros* that get expanded on demand to produce code, while Rose functions defined using `fn` correspond to traditional functions, and must be well-typed. One could define that `sqr` "macro" as a function instead as

```
const sqr = fn([Real], Real, (x) => mul(x, x));
```

where the difference is that the body of this function would then be traced only *once* immediately when it is defined, as opposed to the `sqr` "macro" defined in Listing 1 which gets expanded/traced every time it is called (which in this case happens to only be on line 9). This ability for users to choose between these two ways to define functions is a key feature in the novel design of Rose, and we will see in Section 5 that it is crucial to achieving the design goals for interactive differentiable web applications that we laid out in Section 1.

3.1 Opaque functions

The above example works well using only builtin arithmetic functions, but it's not interactive; let's look at an interactive example that takes advantage of Rose's ability to call predefined JavaScript functions and define custom derivatives for them. The Rose project website² has an interactive widget displaying the local quadratic approximation to the function $(x, y) \mapsto x^y$, allowing a user to drag the point around to see how the shape of the local quadratic approximation shifts; see the full version of this paper [10] for a screenshot. The page also allows the user to modify the mathematical expression defining the function, causing Rose to immediately re-derive the gradient and Hessian, and compile the new function to WebAssembly. For brevity we omit the code to generate the user interface, and instead focus on how one would use Rose to calculate the first and second derivatives used to visualize the quadratic approximation.

Listing 2 shows a Rose program calculating the value, gradient, and Hessian of the power function at $x = 2$ and $y = 3$. Line 2 imports a power function with a custom derivative, as we'll describe shortly. Lines 3–4 define type aliases for \mathbb{R}^2 and $\mathbb{R}^{2 \times 2}$, respectively. Rose types are simply JavaScript values, so type aliases are defined using `const` in the same way as any other JavaScript value.

Recall that the vector-Jacobian product (VJP) introduced in Section 2.1 swaps the domain and codomain from the original function. In addition, JavaScript only allows functions to return one argument. Therefore to take the VJP of a Rose function, that function must have only have one parameter. So, line 5 wraps the `pow` function to take a single vector argument rather than two scalar arguments, allowing it to be passed to Rose's `vjp` function. Just as we discussed in Section 2.1, we compute the gradient by passing in a value of 1.

Lines 7–10 then use the gradient `g` of `f` to compute its Hessian by differentiating once more. Line 8 runs the forward pass for the Hessian just once and saves all necessary intermediate values, after which line 9 runs the backward pass twice with the two basis vectors to compute the full Hessian matrix. Lines 11–14 wrap these three functions into a single function that calls all three and returns the results in a structured form. Finally, line 15 compiles that function to WebAssembly, and line 16 calls it at the point $(2, 3)$.

² <https://rosejs.dev/>

```

1 import { Real, Vec, compile, fn, vjp } from "rose";
2 import { pow } from "./pow.js";
3 const R2 = Vec(2, Real);
4 const R22 = Vec(2, R2);
5 const f = fn([R2], Real, ([x, y]) => pow(x, y));
6 const g = fn([R2], R2, (v) => vjp(f)(v).grad(1));
7 const h = fn([R2], R22, (v) => {
8   const { grad } = vjp(g)(v);
9   return [grad([1, 0]), grad([0, 1])];
10 });
11 const all = fn([Real, Real], { z: Real, g: R2, h: R22 }, (x, y) => {
12   const v = [x, y];
13   return { z: f(v), g: g(v), h: h(v) };
14 });
15 const compiled = await compile(all);
16 console.log(compiled(2, 3));

```

■ Listing 2 An example Rose program.

```

1 import { Dual, Real, add, div, mul, fn, opaque } from "rose";
2 const log = opaque([Real], Real, Math.log);
3 log.jvp = fn([Dual], Dual, ({re:x,du:dx}) => {
4   return { re: log(x), du: div(dx, x) };
5 });
6 export const pow = opaque([Real, Real], Real, Math.pow);
7 pow.jvp = fn([Dual, Dual], Dual, ({re:x,du:dx}, {re:y,du:dy}) => {
8   const z = pow(x, y);
9   const dw = add(mul(dx, div(y, x)), mul(dy, log(x)));
10  return { re: z, du: mul(dw, z) };
11 });

```

■ Listing 3 The contents of `pow.js` defining a differentiable power function.

Listing 3 shows how the `pow` function can be defined to call JavaScript's existing `Math.pow` function. Because Rose cannot see the definition of this `opaque` function, it must be given a definition for its derivative. Lines 3–5 use Rose define the logarithm's dual JVP, which is automatically transposed to produce a VJP as we'll describe in Section 4. Specifically, the signature of this function takes the original `log` function and maps every instance of the `Real` numbers to become the `Dual` numbers we introduced in Section 2.2. In this case, the returned tangent is given by the familiar rule $\frac{d}{dx} \ln x = \frac{1}{x}$ from calculus.

Similarly, lines 6–11 define the power function along with its derivative. Note that, while these two functions use `opaque` to define their bodies, they define their derivatives via `fn`, the same as the Rose functions we discussed in earlier sections. This means that only the first forward derivative needs to be provided. Since the body of this first derivative is transparent to Rose, the reverse derivative and any higher derivatives can be computed automatically.

15:10 Rose: Composable Autodiff for the Interactive Web

```
import { Dual, Real, fn, mul, neg, opaque } from "rose";
const sin = opaque([Real], Real, Math.sin);
const cos = opaque([Real], Real, Math.cos);
sin.jvp = fn([Dual], Dual, ({ re: x, du: dx }) => {
    return { re: sin(x), du: mul(dx, cos(x)) };
});
cos.jvp = fn([Dual], Dual, ({ re: x, du: dx }) => {
    return { re: cos(x), du: mul(dx, neg(sin(x))) };
});
```

■ Listing 4 Definitions of sine and cosine functions with custom derivatives.

```
import { Dual, Real, fn, opaque } from "rose";
const print = opaque([Real], Real, (x) => {
    console.log(x);
    return x;
});
print.jvp = fn([Dual], Dual, (z) => {
    print(z.re);
    return z;
});
```

■ Listing 5 A custom Rose function for print debugging.

3.2 Custom derivatives

Rose lets users define custom derivatives for functions that depend on each other, as in Listing 4. The user can also define their own functions to use with `opaque`. For instance, one might want to define a `print` function for debugging purposes as in Listing 5, but Rose cannot look inside the definition of `print`; by setting `print.jvp`, the programmer can tell Rose that the derivative of this function should similarly perform its side effect and otherwise act like the identity function.

The other situation is when Rose has automatically constructed a derivative for a function, but that derivative is unstable or otherwise exhibits some undesirable property. Rose allows the user to set a custom derivative for any function, not just `opaque` ones. For instance, by default the derivative of the square root function tends to infinity as the argument approaches zero, which causes problems if it is ever called with a zero argument. To prevent this exploding-gradient problem, we sometimes use a square root with a clamped derivative, as in Listing 6.

In all of these examples, notice that the user only needs to specify the JVP, and not the VJP; this is true even if they later decide to use any of these functions in a VJP context, because Rose uses transposition (described in Section 4.3) to automatically construct a VJP from the JVP. A large part of the value of autodiff is that it ensures that the derivative remains in sync with the primal function by construction. Similarly, if we can also assist in keeping the forward-mode and reverse-mode derivatives in sync when one of them must be manually specified, this is a significant benefit for user ergonomics and maintainability.

```

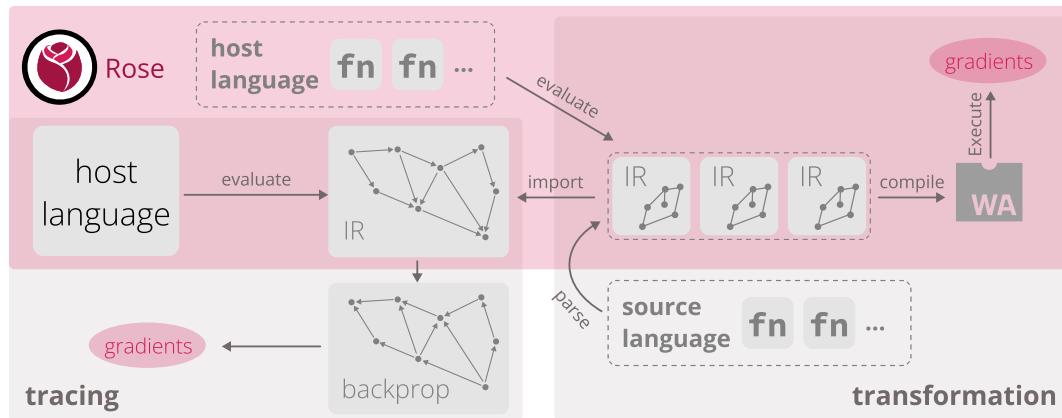
import * as rose from "rose";
import { Dual, Real, div, fn, gt, mul, select } from "rose";

const max = (x: Real, y: Real) =>
  select(gt(x, y), Real, x, y);

const sqrt = fn([Real], Real, (x) => rose.sqrt(x));
sqrt.jvp = fn([Dual], Dual, ({ re: x, du: dx }) => {
  const y = sqrt(x);
  const dy = mul(dx, div(1 / 2, max(1e-5, y)));
  return { re: y, du: dy };
});

```

■ Listing 6 A custom derivative of the square root function to avoid exploding gradients.



■ Figure 1 With **Rose**, the programmer uses the host language for *metaprogramming* like in tracing autodiff, and defines composable *functions* like in **transformation** autodiff.

4 Design

The previous section described a user experience that hints at the design of Rose; this section makes that design explicit. Autodiff frameworks typically fall into the two categories of *tracing* and *transformation*, with some *hybrid* frameworks combining aspects from the two extremes [22]. Rose chooses a specific point in the space of possible hybrid approaches, as diagrammed in Figure 1. To highlight the novel aspects of this design, we will first briefly describe tracing and transformation autodiff; then we will explain how Rose uses parts of both approaches to provide an autodiff engine for the setting described in Section 1.

In tracing autodiff, the programmer writes code in what we call the *host language* (e.g. Python [2, 32]). They use an autodiff library to construct differentiable scalars, vectors, matrices, and other tensors. Each such tensor can be thought of as a single node in a large *computation graph*. Then the programmer calls functions from that autodiff library which take in differentiable tensors and produce more differentiable tensors. Each such function call creates edges in the computation graph from the arguments to the return value. Eventually, a final differentiable scalar value is produced. The programmer calls a special procedure from the autodiff framework, passing in this final scalar value. The autodiff framework traces

backward through the computation graph in reverse topological order, attaching gradient values to every node as it goes. The programmer can then use the autodiff framework to access the gradient value attached to any node as they please.

In transformation autodiff, the programmer writes code in what we will call the *source language* (e.g. Fortran [16] or Julia [20]). The compiler frontend parses and typechecks this source language to convert it to an *intermediate representation (IR)*. This first step typically preserves most of the structure of what the programmer wrote, modulo source formatting. In particular, function definitions and calls in the source text are typically represented as a call graph in an imperative IR, or as lambda terms in a functional IR. Then, the autodiff framework takes in this IR to compute the function the programmer wrote, and emits transformed IR to compute that function along with its gradient. Crucially, this autodiff transformation preserves the asymptotic size of the original program: if the IR representation of the original program has size n , then the size of the transformed program to compute gradients is $O(n)$. Then, the compiler backend converts the IR to an executable binary like normal, which can be run to compute the desired gradients.

Figure 1 shows how Rose combines these two approaches. Like tracing, Rose lets the programmer write in a host language they are familiar with: JavaScript, in this case. And like tracing, the programmer is free to use all the features of the host language to describe the shape of their computation. But unlike tracing, and more like transformation, Rose also allows the programmer to explicitly define multiple functions that can be composed together to form a larger computation. Unlike transformation, the programmer’s code does not get directly parsed and typechecked to produce the IR; the IR is instead produced by symbolic evaluation like in tracing. But like transformation, the IR can include control flow and function calls, which get explicitly transformed by autodiff rather than being effectively erased as in tracing. And like transformation, the resulting differentiated IR is compiled to WebAssembly [15] that can then be repeatedly executed to compute gradients for the same program.

By restricting our IR to not allow recursive functions, we are able to use a compilation strategy similar to destination-passing style [40] to avoid the cost of sophisticated memory management, increasing performance. This strategy not to deallocate memory while computing gradients is justified by the way that reverse-mode autodiff generally needs to retain intermediate values, as described in Section 4.3. Importantly, while Rose IR does not allow recursion, the programmer can freely express recursive computations by using the host language for metaprogramming, as we will later discuss in Sections 4.4 and 5.4.2.

In the following subsections, we will discuss the Rose IR at a theoretical level and explain the autodiff and transposition [39] program transformations which we use to compute gradients; then in Section 4.4 we will step back again to discuss how the programmer interacts with this IR indirectly through Rose as a library. All inference rules can be found in the full version of this paper [10].

4.1 Rose intermediate representation

Figure 2 shows the abstract syntax for the Rose IR. It is a first-order functional language with non-mutable array types, and a “reference” or “accumulator” type constructor [33], written $\&\tau$. The full version of this paper [10] gives the typing rules for the Rose IR. These are all fairly standard, except for the rules for type constraints κ . We will explain these less common features of the language in the context of a specific example. Section 4.3 will demonstrate the need for accumulators in more detail, but at a high level, they arise naturally because the backward pass of reverse-mode autodiff essentially runs the program backward in

```

 $m, n \in \mathbb{Z}_{\geq 0}$ 
 $c \in \mathbb{R}$ 
 $\kappa ::= \text{Type} \mid \text{Value} \mid \text{Index}$ 
 $\tau ::= t \mid () \mid \text{Bool} \mid \text{Real} \mid n \mid \&\tau \mid [\tau]\tau \mid (\tau, \tau) \mid \langle t: \kappa \rangle(\underline{\tau}) \rightarrow \tau$ 
 $\ominus ::= \neg \mid - \mid \text{abs} \mid \text{sgn} \mid \text{ceil} \mid \text{floor} \mid \text{trunc} \mid \text{sqrt}$ 
 $\oplus ::= \wedge \mid \vee \mid \text{iff} \mid \text{xor} \mid \neq \mid < \mid \leq \mid = \mid > \mid \geq \mid + \mid - \mid \times \mid \div$ 
 $e ::= () \mid \text{true} \mid \text{false} \mid c \mid n \mid [\underline{x}] \mid (x, x) \mid \ominus x \mid x \oplus x \mid x ? x : x \mid x += x$ 
 $\mid x[x] \mid \text{fst } x \mid \text{snd } x \mid \&x[x] \mid \&\text{fst } x \mid \&\text{snd } x \mid f\langle \underline{\tau} \rangle(\underline{x}) \mid [\text{for } x: \tau, b]$ 
 $\mid \text{accum } x \text{ from } x \text{ in } b$ 
 $b ::= x \mid \text{let } x: \tau = e \text{ in } b$ 
 $d ::= \text{def } f\langle \underline{t}: \kappa \rangle(\underline{x}: \tau) : \tau = b$ 

```

Figure 2 Abstract syntax for Rose IR.

```

1 def sum<n: Index>(v: [n]Real): Real =
2   let z: Real = 0.0 in
3   let t: (Real, [n]()) =
4     accum a from z in
5       [for i: n,
6        let x: Real = v[i] in
7        let u: () = a += x in
8        u
9      ]
10     in
11   let y: Real = fst t in
12   y

```

Listing 7 A Rose IR function to compute the sum of a vector of real numbers.

time, so reads become accumulation, and writes become reads. Having a type which restricts mutation in this way makes it easier to ensure correctness without introducing additional data dependencies that hinder compiler optimization.

Rose users write JavaScript, so prior examples have been written in JavaScript. However, in this subsection, and Sections 4.2 and 4.3, we describe the IR and so the examples are written in Rose IR. To make this distinction clear, we will format IR examples differently by putting them inside grey boxes.

Listing 7 computes the sum of a vector of real numbers. Line 1 says that the function is generic over the size of the array, where the size is represented as a type n with the `Index` constraint. The three type constraints in Rose IR are ordered as a hierarchy `Index` $<$ `Value` $<$ `Type`. Semantically, `Type` refers to any type that can be stored in a variable; the only types that don't satisfy `Type` are function types, since Rose IR is first-order. `Value` is contrasted with reference types: that is, types $\tau : \text{Type}$ satisfy the `Value` constraint unless they are of the form $\tau = \&\tau'$. Only `Value` types can be used in other type constructors, so for example, a reference cannot be stored as an element of an array. Finally, the `Index` constraint marks types that can be used as the index type of an array; the only Rose IR types that satisfy this constraint are those of the form $n \in \mathbb{N}$, which correspond to the value set $\{0, \dots, n - 1\}$.

Line 2 defines a local called `z` of type `Real` with the value `0.0`. This is used by the `accum` block on lines 4 to 10. This block serves as the scope for the variable `a` of type `&Real`, because `z` is of type `Real`. The variable `a` is in scope for lines 5 to 9 and goes out of scope on line 10. As mentioned above, this is an *accumulator* type: it represents a container holding a value of type `Real`, but that value cannot be read, and can only be accumulated. In other words, there is no operation of type `&Real → Real`. But, given a value of type `Real`, one can use the `+=` operation to accumulate into the value contained in `a`.

With this accumulator in hand, lines 5 to 9 execute. These lines are an *array constructor* with index type `n`, as shown on line 5. The value type of this array constructor is the unit type `()` so its resulting array type `[n]()` holds no data; its sole importance comes from the side effect it performs on line 7. The result is that every element of `v` gets accumulated into the value of `a`, so after this `for` block executes, the inner value of `a` is equal to the sum of all the values from `v`. Again, though, `a` cannot actually be read.

Finally, after the body of the `accum` block executes, it returns the inner value from `a`, along with the value that was returned from the `accum` block body itself, which is of type `[n]()` as mentioned before. These two items are packaged together into a tuple `t` of type `(Real, [n]())`. Because no accumulator type can be part of a tuple, this prevents `a` itself from escaping from the `accum` body, so it is guaranteed to be inaccessible after the `accum` block executes. Thus, every accumulator of type `&τ` starts as accumulate-only, and then when it goes out of scope, its value is guaranteed to be inaccessible except as a “decayed” read-only value of type `τ`. These semantics may seem unintuitive, but they turn out to perfectly model the mapping from forward-mode to reverse-mode autodiff described in Section 4.3.

That function definition was quite verbose, as it strictly adhered to the syntax from Figure 2 for clarity. In the remainder of this section, we will allow ourselves syntactic sugar to write expressions in places where the strict syntax requires variable names, with the understanding that these could be desugared by introducing intermediate `let` bindings:

```
def sum<n: Index>(v: [n]Real): Real =
  fst (accum a from 0.0 in [for i: n, a += v[i]])
```

4.2 Forward-mode autodiff

As we showed in Section 2.2, the forward-mode JVP can often be easier to specify than the reverse-mode VJP, which is why we allow the user to specify custom derivatives using the JVP as described in Section 3.2. So, we decompose reverse-mode autodiff into two parts [39], first applying forward-mode autodiff as we will describe shortly, and then applying a second transformation which we will describe in Section 4.3. The full version of this paper [10] lists inference rules for forward-mode autodiff of Rose IR. Specifically, these rules can be used to transform a function f into a function f' that computes the *dual JVP* of f , where the dual numbers are represented in Rose IR by the tuple type `Dual = (Real, Real)`.

Consider this function, which assumes that `sin : <>(Real) → Real` already exists:

```
def f(u: Real): Real =
  let v: Real = sin(u) in
  let w: Real = -v in
  w
```

We assume that we are already given a dual JVP for `sin`. For instance, if the function `cos : <>(Real) → Real` also exists, then `JVPsin : <>(Dual) → Dual` might be:

```
def jvp_sin((x, dx): Dual): Dual = (sin(x), dx * cos(x))
```

Then, by applying the inference rules we have laid out, we get $\overline{\text{jvp}}_f : \langle\rangle(\text{Dual}) \rightarrow \text{Dual}$:

```
def jvp_f(u: Dual): Dual =
  let v: Dual = jvp_sin(u) in
  let (v_re, v_du) = v in
  let w: Dual = (-v_re, -v_du) in
  w
```

All the rules for forward-mode autodiff are straightforward and quite standard, so we will not dwell on them here. Now we move on to the more complicated transformation, which maps from forward-mode autodiff to reverse-mode autodiff.

4.3 Transposition

The name “transposition” comes from the fact that the JVP and VJP can be thought of as transposes of each other, in the sense of transposing a matrix. Recall that, for $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ and $x \in \mathbb{R}^n$, we have

$$\begin{aligned} \text{jvp}_f^x : \mathbb{R}^n &\rightarrow \mathbb{R}^m & \text{vjp}_f^x : \mathbb{R}^{1 \times m} &\rightarrow \mathbb{R}^{1 \times n} \\ \text{jvp}_f^x(\dot{x}) &= \mathbf{J}_f(x)\dot{x} & \text{vjp}_f^x(\ddot{y}) &= \ddot{y}\mathbf{J}_f(x) \end{aligned}$$

which both make use of the Jacobian matrix $\mathbf{J}_f(x) : \mathbb{R}^{m \times n}$. So, $\mathbf{J}_f(x)$ is the matrix of the linear transformation jvp_f^x . But then, observe that $\mathbf{J}_f(x)^\top : \mathbb{R}^{n \times m}$ is the matrix of the linear transformation

$$\begin{aligned} \mathbb{R}^m &\rightarrow \mathbb{R}^n \\ \dot{y} &\mapsto \mathbf{J}_f(x)^\top \dot{y} = (\dot{y}^\top \mathbf{J}_f(x))^\top = \text{vjp}_f^x(\dot{y}^\top)^\top. \end{aligned}$$

If we had an explicit dense representation of \mathbf{J}_f then it would be trivial to transpose. But we don’t; instead, we have an implicit, potentially sparse, representation of \mathbf{J}_f in the form of the dual JVP function $\overline{\text{jvp}}_f$. The full version of this paper [10] lists inference rules to transpose the linear derivative represented by the dual JVP in Rose IR. This transformation is considerably more complicated than the initial transformation to do forward-mode autodiff. We will walk through this transformation using the example from Section 4.2, leaving a more systematic exposition to the prior literature [33, 39] on which our approach was based.

Now, back to the example. By strictly following the inference rules we have laid out, we end up with Listing 8. As is common with program transformations like this, the resulting code is highly redundant; via a straightforward optimization pass that we omit here for brevity, we obtain the following:

```
def fwd_f((u, _): Dual): (Dual, Tape_sin) =
  let ((v, _), t) = fwd_sin((u, 0)) in
  let w = -v in
  ((w, 0), t)
def bwd_f(ddu: &Dual, (_, dw): Dual, t: Tape_sin): () =
  let dv = -dw in
  bwd_sin(ddu, (0, dv), t)
```

This example hints at the fact that the infinitesimal part is always zero for dual numbers representing primal values, and the real part is always zero for dual numbers representing adjoint values. Thus, in our actual system, in the aforementioned optimization pass we also translate the `Dual` type back to `Real`, essentially reversing our replacement of `Real` with `Dual` from Section 4.2:

15:16 Rose: Composable Autodiff for the Interactive Web

```

type Tf = (Dual, (Dual, (Tape_sin, (Real, (Real, (Dual, ()))))))
def fwd_f(u: Dual): (Dual, Tf) =
  let (v, t0) = fwd_sin(u) in
  let (v_re, v_du) = v in
  let w_re = -v_re in
  let w_du = 0 in
  let w = (w_re, v_re) in
  (w, (v, (t0, (w_re, (w_du, (w, ()))))))
def bwd_f(ddu: &Dual, dw0: Dual, t: Tf): () =
  let (v, (t0, t1)) = t in
  let (dv, ()) = accum ddv from v in (
    let v_re = fst v in
    let ddv_re = &fst ddv in
    let v_du = snd v in
    let ddv_du = &snd ddv in
    let (w_re, t2) = t1 in
    let (dw_re, ()) = accum ddw_re from w_re in (
      let (w_du, t3) = t2 in
      let (dw_du, ()) = accum ddw_du from w_du in (
        let (w, t4) = t3 in
        let (dw1, ()) = accum ddw from w in (
          ddw += dw0
        ) in
        let (dw1_re, dw1_du) = dw1 in
        ddw_re += dw1_re ;
        ddw_du += dw1_du
      ) in
      ddv_du += -dw_du
    ) in
    ()
  ) in
  bwd_sin(ddu, dv, t0)

```

■ Listing 8 Strict transposition of the function f from Section 4.2.

```

def fwd_f(u: Real): (Real, Tape_sin) =
  let (v, t) = fwd_sin(u) in
  let w = -v in
  (w, t)
def bwd_f(ddu: &Real, dw: Real, t: Tape_sin): () =
  let dv = -dw in
  bwd_sin(ddu, dv, t)

```

Thus concludes the transposition of f. Similarly we can also transpose jvp_sin:

```

def fwd_sin(x: Real): (Real, Real) =
  (sin(x), cos(x))
def bwd_sin(ddx: &Real, dy: Real, z: Tape_sin): () =
  ddx += dy * z

```

```

import { Real, add, div, fn } from "rose";
const f = (x) =>
  fn([Real], Real, (y) => div(x, y));
const g = (y) =>
  fn([Real], Real, (x) => div(x, y));
let [f0, f1, f2] = [f(5), f(7), f(5)];
fn([Real], Real, (z) => add(f1(z), f1(z)));

```

```

def f0(y: Real): Real =
  5 / y
def f1(x: Real): Real =
  7 / y
def f2(x: Real): Real =
  5 / y
def h(z: Real): Real =
  f1(z) + f1(z)

```

Figure 3 JavaScript code (left) that produces Rose IR (right) when evaluated.

4.4 Metaprogramming

We have described the right-hand side of Figure 1; now all that remains in this section is to describe the “evaluate” step on the left-hand side. Figure 3 shows a simple example of how evaluating JavaScript code produces Rose IR. In this example, we have two JavaScript functions `f` and `g`, each of which uses `fn` to construct and return a Rose function when called. We call `f` three times and never call `g`, so three Rose IR functions resulted from the single instance of `fn` in the source of `f`, and zero Rose IR functions resulted from the instance of `fn` in the source of `g`. Then, we call `fn` one more time, and in the body of that `fn`, we call `f1` twice. Note that those calls to `f1` remain in the resulting IR; Rose does not inline calls, in contrast to standard tracing autodiff frameworks.

5 Evaluation

As discussed in Section 4, Rose is characterized by three primary design choices:

- D1 Allow users to define and compose custom functions using `fn`.
- D2 Use program transformation to compile to WebAssembly.
- D3 Use tracing to allow metaprogramming in JavaScript.

These design choices are motivated by Rose’s role as a toolkit for building interactive, differentiable web applications. The dynamic bandwidth- and latency-constrained environment of web browsers poses significant constraints on the size and speed of Rose. In addition to good performance, Rose also needs to be expressive and flexible enough for the end user to build web applications in a myriad of domains.

In this section, we evaluate Rose both quantitatively and qualitatively by integrating Rose into three web-based applications for diagramming, physical simulation, and reinforcement learning (Section 5.1). In subsequent subsections, we report on Rose’s performance (Sections 5.2 and 5.3) and discuss qualitative observations of how Rose’s design choices impact its expressiveness and flexibility (Section 5.4).

5.1 Benchmark and applications

To the best of our knowledge, there are no benchmark suites for evaluating autodiff performance generally [41], let alone web-based autodiff of scalar programs. Further, measuring performance on just a benchmark would limit our ability to qualitatively evaluate the expressiveness of Rose in real-world applications. Therefore, we opted to find applications in which autodiff plays a central role, and re-implement the autodiff module or the entire application using Rose. We believe this approach gives us better ecological validity (i.e. the realism of our evaluation setup) and potentially leads to a rich source of examples. Our

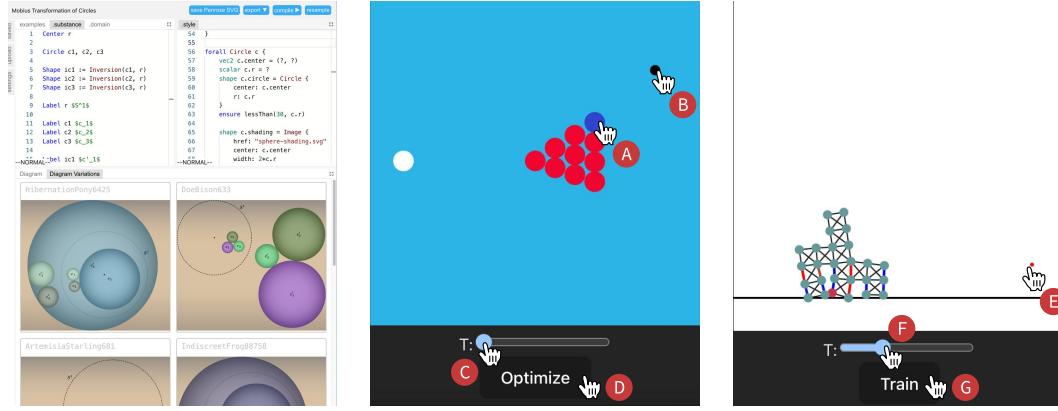


Figure 4 Three web-based applications re-implemented with Rose. **Left:** the Penrose IDE. **Middle:** billiards simulator that optimizes **D** for cue ball angle and speed such that the object ball **A** reaches the target **B**. **Right:** mass-spring robot controlled by a neural net trained **G** with a designated goal **D**. Both simulations can be replayed by dragging the sliders at any point **D** and **G**.

search resulted in two frameworks that are rich sources of applications and benchmarks: **Penrose** [54], a web-based diagramming framework; and **Taichi** [19], a Python library for high-performance parallel programming. The two frameworks also have sufficiently different settings that together they illustrate the flexibility of Rose.

Penrose allows the user to specify a diagram by constructing a custom numerical optimization problem in a DSL called Style, then runs a numerical solver to rearrange the shapes in the diagram until it finds a local minimum. Optimizing the layout of these diagrams involves defining and differentiating a wide range of mathematical operations on scalars, from simple operations like finding the distance between points to more sophisticated calculations like Minkowski addition, KL divergence, and silhouette points. The main application of Penrose is a web-based IDE (Figure 4, left), where users live-edit programs to produce layout-optimized diagrams. Importantly, the framework uses TensorFlow.js for autodiff and ships with 173 “registry” diagrams for performance testing, each of which was compiled to a unique differentiable computation. Therefore, we deemed it as a suitable target for performance comparison with TensorFlow.js, the closest baseline to which we can compare Rose’s performance. This set of registry diagrams is quite diverse, comprising a total of 97 unique Style programs which are preprocessed by the Penrose compiler frontend before being passed to Rose.

Many applications of Taichi involve differentiable programming and DiffTaichi [18] presents a few example Python applications that combines physical simulation and neural networks. We used Rose to implement and augment two such differentiable simulations from Taichi: **billiards** and **robot** (Figure 4, middle and right). The **billiards** example is a differentiable simulation of pool combination shots. The program simulates rigid body collisions between a cue ball and object balls. Leveraging the differentiability of the simulation, a gradient descent optimizer solves for the initial position and velocity of the cue ball to send a designated object ball to a target position. The **robot** example simulates a robot made of a mass-spring system, where springs are actuated to move the robot towards a goal position. A neural network controller is trained on simulator gradients to update the spring actuation magnitude over time. In both cases, the simulation is run to completion, remembering intermediate computations along the way, and then autodiff is used to run back through the simulation in reverse to compute gradients for the initial state.

All three applications were implemented using Rose’s JavaScript binding. They all run in major browsers such as Safari and Chrome. To showcase the benefits of running in the web browser, we added interactive features to the Taichi applications (Figure 4). For instance, the Taichi version of `billiards` is a command-line application that outputs a series of static images based on hard-coded parameters for the choice of the object ball and goal position. The Rose version allows the user to interactively explore the simulator by selecting the object ball (Figure 4(A)), moving the goal position (Figure 4(B)), optimizing the cue ball position (Figure 4(D)), and re-playing the simulation (Figure 4(C)).

5.2 Size

Similar to TensorFlow.js, Rose is a client-side JavaScript package that is typically bundled with the rest of a web application and delivered over the internet. To run in web browsers, Rose needs to be comparable or smaller than similar packages such as TensorFlow.js. As a baseline, a common TensorFlow.js distribution, `@tensorflow/tfjs-core` version 4.18.0 (latest at time of writing), is 479.98 kB after minification (85.88 kB after gzip). We publish Rose via the npm package `rose`,³ which is at version 0.4.10 at time of writing. The Rose WebAssembly binary size [5] is 168.51 kB (63.97 kB after gzip), and the JavaScript wrapper layer is 31.77 kB after minification (8.74 kB after gzip). For a more extreme comparison, there are projects that package heavy compiler infrastructure like LLVM to WebAssembly [47], but those produce binaries on the order of a hundred megabytes, causing unacceptable load times for end users. Another reference point is the Taichi package on PyPI, which is about 50–80 megabytes depending on the platform. It is unclear how difficult it would be to package Taichi to run in the browser using Pyodide [37], which is 6.4 megabytes and whose authors claim to be 3×–5× slower than native Python.

5.3 Performance

To compare with the TensorFlow.js baseline for execution performance, we replaced the Penrose TensorFlow.js-based autodiff engine with one written in Rose and ran both versions on the benchmark of 173 diagrams (Section 5.1). We measured the amount of time it took for each autodiff engine to perform any necessary compilation, plus the time taken by the Penrose L-BFGS [27] optimization engine to converge on each diagram. We specifically include the time it takes for Rose to do autodiff, transposition, and WebAssembly compilation, despite the fact that TensorFlow.js does not have an analogous compilation step. On the surface this puts Rose at a disadvantage, but fast compilation time is essential when constructing Rose functions dynamically in a user-facing web application, as Penrose does.

Figure 5 shows the results.⁴ The quartiles for the ratio of TensorFlow.js optimization time to Rose optimization time were 37×, 173×, and 598×. These results show that Rose provides an enormous advantage over TensorFlow.js (the state-of-the-art for autodiff on the web) for scalar programs like those found in Penrose diagrams. Because these numbers include both compile time and optimization time, the results demonstrate the end-to-end performance of Rose.

³ <https://www.npmjs.com/package/rose>

⁴ All numbers we report in this section were measured in the V8 JavaScript engine (used in both Chrome and Node.js) on a 2020 MacBook Pro with M1 chip.

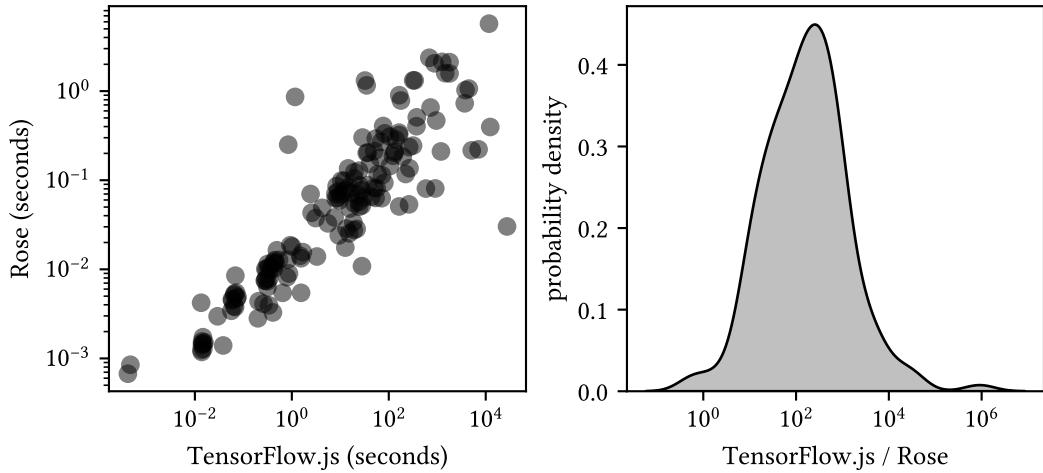


Figure 5 Left: Log-log scatterplot of Penrose diagram optimization time with TensorFlow.js versus Rose. **Right:** Log-scale kernel density estimate (KDE) plot of the optimization time of TensorFlow.js to Rose.

We omitted 10 of the 173 diagrams from our data analysis:

- **9 NaN failures:** Penrose aborts if it detects a “not-a-number” (NaN) value in the gradient as it is optimizing. This occurred in the TensorFlow.js version of Penrose for nine diagrams. *The Rose version of Penrose did not encounter NaNs for these programs.*
- **1 timeout:** For one diagram, we stopped the TensorFlow.js version of Penrose after it had run for over 24 hours. *The Rose version of Penrose took 42 milliseconds to compile and optimize this diagram.*

Tensorflow.js runs on both CPU and GPU. We used the “cpu” backend in our comparison because we found that, for scalar programs, it was faster than their GPU backend. To double-check this, we took the 88 diagrams (over half) that were quickest to run with TensorFlow.js, and also ran them with `@tensorflow/tfjs-node` and `@tensorflow/tfjs-node-gpu`, which they claim are faster than the “cpu” backend. We found that the Node backend is *79% slower* (median ratio) than the “cpu” backend, and the Node GPU backend is *75% slower* (median ratio) than the “cpu” backend. Also, those backends are unable to run in a browser, unlike the “cpu” backend, so they would be inappropriate for a direct comparison to Rose.

As we will discuss in Section 5.4.1, Rose’s ability to define separate functions in a graph (rather than just a single graph of scalar or tensor values) is crucial to producing small enough WebAssembly binaries to feed to the browser. To investigate whether WebAssembly brought significant performance gains in the first place to be worth facing that challenge, we compared against a modified version of Rose which emits JavaScript code instead of WebAssembly. This experiment gave quartile slowdowns of 10%, 49%, and 100% for optimization of Penrose diagrams, showing that WebAssembly provides a significant advantage over JavaScript as a compilation target for Rose.

For the two Taichi applications (Figure 4, middle and right), we compare the running time of training/optimizing with the Python command-line counterparts. On average (of 10 runs), Rose is on par with the native performance of the Python versions: for the default initial condition in `billiards`, Rose completed the optimization in $22.7s \pm 0.2s$ while Taichi completed it in $20.6s \pm 0.8s$; for the default condition in `robot`, Rose finished 100 iterations of learning in $32.1s \pm 0.2s$ while Taichi took $31.3s \pm 1.1s$.

In the Penrose IDE (Figure 4, left), the main interaction that will trigger differentiation is compiling the DSL to diagrams. As reported in the previous section, Rose-based Penrose is significantly faster than the TensorFlow.js version, often leading to visible reduction in diagram layout optimization time in the user interface.

5.4 Qualitative observations

Our implementation effort to port both Penrose and Taichi applications to Rose spanned thousands of lines of code, including replacing the Penrose autodiff engine and function library and rewriting both `billiards` and `robot` for scratch. In this process, we have written a wide variety of differentiable programs using Rose, and had a chance to observe how Rose’s main design choices (D1, D2, D3) impact the way programs are written. In this section, we report on our qualitative observations using Rose in these real-world settings, highlighting how these design choices interact with each other to form a coherent system.

5.4.1 Writing scalar programs as composable functions

The original versions of Penrose, `billiards`, and `robot` are naturally written as scalar programs. In Penrose, `bboxCircle` (line 10 of Listing 9) computes the bounding box by performing arithmetic on scalar values for the center and radius of a circle. In Taichi, both `billiards` and `robot` involve hand-crafted scalar programs for differentiable simulations. For instance, `apply_spring_force` (Figure 6) loops through individual springs in the robot, computing the force on the spring based on scalar-valued parameters, and scatter forces to end points of springs.

Because Rose is designed for writing scalar programs, translating both Penrose and Taichi source programs to Rose is straightforward and largely preserves the structures of the programs. For instance, when translating the Python programs from Taichi into TypeScript and Rose, as shown in Figure 6, Taichi kernels can be translated one-to-one to Rose functions.

Reflecting on the design choices, the combination of transformation to WebAssembly (D2) and the basic building block of composable functions (D1) gives the user both performance gains and an ergonomic programming interface. In the case of Taichi, the Rose abstraction of `fn` is not only useful for one-to-one translation from Taichi, but also necessary for running the simulator in browsers. Major WebAssembly engines have limits on WebAssembly binary size and on the number of local variables in each function. While it is possible to encapsulate much of the simulation code of `billiards` and `robot` in bigger JavaScript functions, the compiled size and local counts of these functions would quickly exceed these limits and would not run in the browser. Therefore, segmenting the source into functional units of `fns` effectively reduces the size of emitted WebAssembly functions and modules, avoiding these errors and reducing compile times.

5.4.2 Metaprogramming and function dynamism

The Rose IR is designed to be performant and easy to compile to WebAssembly (D2) and therefore has limited expressiveness (Section 4). Metaprogramming using JavaScript enables the user to dynamically generate complex computation graphs that are impossible to specify with the Rose IR alone (D3). For instance, the `bboxGroup` function in Listing 9 computes the bounding box of a `Group` in Penrose, a recursive collection of shapes. For non-collection shape types such as `Circle`, we ported the TensorFlow.js implementation to Rose easily, e.g. `bboxCircle`. However, `bboxGroup` needs to recurse over the `Group` data structure to find out

```

@ti.kernel
def apply_spring_force(t: ti.i32):
    for i in range(n_springs):
        a = spring_anchor_a[i]
        b = spring_anchor_b[i]
        pos_a = x[t, a]
        pos_b = x[t, b]
        dist = pos_a - pos_b
        length = dist.norm() + 1e-4
        target_length = spring_length[i] *
            (1.0 + spring_actuation[i] * act[t, i])
        impulse = dt * (length - target_length) *
            spring_stiffness[i] / length * dist
        ti.atomic_add(v_inc[t + 1, a], -impulse)
        ti.atomic_add(v_inc[t + 1, b], impulse)

    const apply_spring_force = fn(
        [Objects, Act], Objects, (x, act) => {
            const v_inc = [];
            for (let i = 0; i < n_objects; i++)
                v_inc.push([0, 0]);
            for (let i = 0; i < n_springs; i++) {
                const spring = robot.springs[i];
                const a = spring.object1;
                const b = spring.object2;
                const pos_a = x[a];
                const pos_b = x[b];
                const dist = vsub2(pos_a, pos_b);
                const length = add(norm(dist), 1e-4);
                const target_length = mul(spring.length,
                    add(1, mul(act[i], spring.actuation)));
                const impulse =
                    vmul(div(mul(dt * spring.stiffness,
                        sub(length, target_length)),
                        length),
                        dist);
                v_inc[a] = vsub2(v_inc[a], impulse);
                v_inc[b] = vadd2(v_inc[b], impulse);
            }
            return v_inc;
        });

```

Figure 6 A function that applies spring actuation on the mass-spring robot model in the `robot` example, written in Taichi (**Left**) and Rose (**Right**). The translation from Taichi to Rose is straightforward.

the bounding boxes of individual shapes before aggregating them into the final bounding box. This requires conditional dispatch of (1) Rose functions based on a discrete tag (`shape.kind`) and (2) recursive calls to `bboxGroup` to handle nested groups.

Figure 7 shows an example of calling `bboxGroup` on nested groups of shapes. The diagram in Figure 7 (left) has 1 group containing the whole diagram, and 3 subgroups of molecules that contain shapes such as `Text` and `Circle`. Figure 7 shows how `bboxGroup` uses JavaScript language features to compose Rose functions into a computation graph, denoting JavaScript constructs in gray and Rose functions in red. First, for each member shape, we `switch` on `shape.kind` to determine whether to (a) call individual Rose bounding box functions like `bboxCircle` or (b) recurse to call `bboxGroup`. Then, after all the child bounding boxes are computed, we use JavaScript `map` and `reduce` to aggregate via Rose `min` and `max` functions.

In the case of Penrose, metaprogramming actually helped us reduce the lines of code to refactor, because many plain JavaScript functions can stay the same and we only had to refactor functions that involve actual computation. We also speculate that by reducing the size of Rose-specific constructs, new users can learn a smaller API easier and experience a smoother learning curve.

6 Related work

Autodiff first started being seriously studied a few decades ago [48], with Griewank and Walther’s book [14] consolidating research on the topic up until its publication. Some tools were developed, such as Tapenade [16] which operates over C and Fortran. The machine learning community developed an interest in autodiff over the past decade, resulting in popular tools including TensorFlow [2], PyTorch [32], and JAX [12] as mentioned in Section 1. JAX is particularly interesting because in a way it blends together tracing and transformation like we do here, but unlike Rose, JAX is not scalar-friendly and does not allow the programmer to explicitly define functions to serve as boundaries for tracing. Other autodiff systems include Zygote [21] for Julia, and Enzyme [29, 30, 31] for LLVM IR [26]. For graphics programming, $\text{A}\delta$ [53] and Dr.Jit [22] can be used to differentiate shaders.

```

1  const bboxGroup = (shapes) => {
2    const bboxes = shapes.map(bbox);
3    const left = bboxes.map((b) => b.left).reduce(min);
4    const right = bboxes.map((b) => b.right).reduce(max);
5    const bottom = bboxes.map((b) => b.bottom).reduce(min);
6    const top = bboxes.map((b) => b.top).reduce(max);
7    return { left, right, bottom, top };
8  };
9
10 const bboxCircle = fn([Circle], Rectangle,
11   ({ center: [x, y], radius: r }) => {
12     const left = sub(x, r);
13     const right = add(x, r);
14     const bottom = sub(y, r);
15     const top = add(y, r);
16     return { left, right, bottom, top };
17   },
18 );
19
20 const bbox = (shape) => {
21   switch (shape.kind) {
22     case "Rectangle": return shape.value;
23     case "Circle": return bboxCircle(shape.value);
24     case "Group": return bboxGroup(shape.value);
25   }
26 };

```

Listing 9 Examples of JavaScript metaprogramming to construct Rose functions for recursive data structures.

The programming languages community has also taken an interest in autodiff [36], producing proofs of correctness [1], program transformations for SSA [20] and CPS [51], and more recently, reformulations of reverse-mode autodiff in terms of dual numbers [44], as well as a new autodiff formulation called CHAD [49, 45]. Some work also attempts to bridge the gap between programming language theory and the machine learning domain by facilitating automatic parallelization [6, 33]. The latter work also resulted in a formalization of function transposition [39] which directly inspired the low-level design of autodiff in Rose.

Rose supports higher-order derivatives because its core IR is closed under differentiation and transposition. A more sophisticated approach we don't explore here would be derivative towers [23, 35], sometimes called “Taylor towers” because they use Taylor expansions instead of the chain rule. We would be interested to see how derivative towers can be combined with our approach in future work, while avoiding the pitfalls of perturbation confusion [42, 28]. Another optimization that becomes crucial when scaling up autodiff is checkpointing, which cuts down drastically on memory requirements; for instance, while the semantics of Rose can in general result in keeping around arbitrarily many intermediate results, a recursive “divide-and-conquer” checkpointing scheme [43] reduces the memory impact of reverse-mode autodiff to a logarithmic factor; the cost, though, is that the asymptotic running time would also suffer a logarithmic factor.

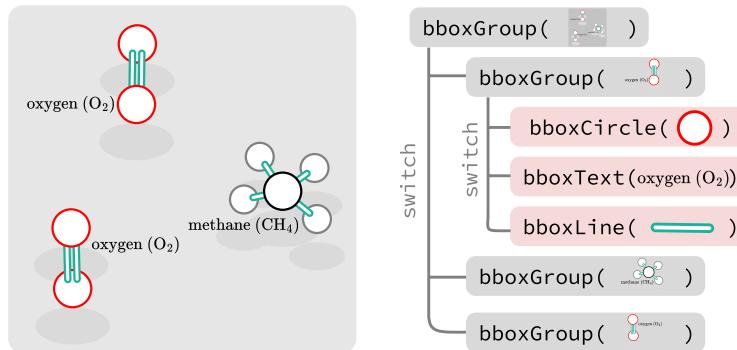


Figure 7 In Penrose, we used JavaScript to programmatically generate Rose functions. **Left:** a figure comprised of a top-level group containing all molecules and sub-groups for each molecule. **Right:** the `bboxGroup` function conditionally generates Rose functions or recursively calls itself based on the shape type.

7 Conclusion and future work

This paper introduced Rose, an embedded domain-specific language for automatic differentiation of interactive programs on the web, which blends together the two primary autodiff techniques of tracing and transformation. Currently Rose targets WebAssembly, which runs on the CPU; as we showed in Section 5.3, this already provides an enormous performance advantage for scalar programs when compared to the state-of-the-art for autodiff on the web. In future work, we would also like to pursue further performance gains by implementing a backend that targets WebGPU [24]. We have already laid the groundwork for this by drawing inspiration for the Rose IR from Dex [33] to be friendly to automatic parallelization, such as the `for` construct and accumulate-only reference types. In general, we plan to continue this line of work to open up new modes of differentiable interactivity.

References

- 1 Martín Abadi and Gordon D. Plotkin. A simple differentiable programming language. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi:[10.1145/3371106](https://doi.org/10.1145/3371106).
- 2 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, Dan Mane, 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 Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous distributed systems, 2016. arXiv:[1603.04467](https://arxiv.org/abs/1603.04467).
- 3 Anthony Anjorin, Li-yao Xia, and Vadim Zaytsev. Bidirectional transformations wiki, 2011. URL: <http://bx-community.wikidot.com/>.
- 4 Francis J. Anscombe. Graphs in statistical analysis. *The American Statistician*, 27(1):17–21, 1973.
- 5 Hudson Ayers, Evan Laufer, Paul Mure, Jaehyeon Park, Eduardo Rodelo, Thea Rossman, Andrey Pronin, Philip Levis, and Johnathan Van Why. Tighten Rust’s belt: Shrinking embedded Rust binaries. In *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2022, pages 121–132, New York, NY, USA, 2022. Association for Computing Machinery. doi:[10.1145/3519941.3535075](https://doi.org/10.1145/3519941.3535075).

- 6 Gilbert Bernstein, Michael Mara, Tzu-Mao Li, Dougal Maclaurin, and Jonathan Ragan-Kelley. Differentiating a tensor language, 2020. [arXiv:2008.11256](https://arxiv.org/abs/2008.11256).
- 7 Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing, 2012. [arXiv:1209.5145](https://arxiv.org/abs/1209.5145).
- 8 Dan Cascaval, Mira Shalah, Phillip Quinn, Rastislav Bodik, Maneesh Agrawala, and Adriana Schulz. Differentiable 3D CAD programs for bidirectional editing. *Computer Graphics Forum*, 41(2):309–323, 2022. doi:[10.1111/cgf.14476](https://doi.org/10.1111/cgf.14476).
- 9 Bartosz Ciechanowski, 2014. URL: <https://ciechanow.ski/>.
- 10 Sam Estep, Wode Ni, Raven Rothkopf, and Joshua Sunshine. Rose: Composable autodiff for the interactive web, 2024. [arXiv:2402.17743](https://arxiv.org/abs/2402.17743).
- 11 Figma, Inc. Figma, 2016. URL: <https://figma.com/>.
- 12 Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, 4(9), 2018.
- 13 Google LLC. Google Slides, 2006. URL: <https://google.com/slides>.
- 14 Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, 2008.
- 15 Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the Web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 185–200, New York, NY, USA, 2017. Association for Computing Machinery. doi:[10.1145/3062341.3062363](https://doi.org/10.1145/3062341.3062363).
- 16 Laurent Hascoet and Valérie Pascual. The Tapenade automatic differentiation tool: Principles, model, and specification. *ACM Trans. Math. Softw.*, 39(3), May 2013. doi:[10.1145/2450153.2450158](https://doi.org/10.1145/2450153.2450158).
- 17 Brian Hempel, Justin Lubin, and Ravi Chugh. Sketch-n-Sketch: Output-directed programming for SVG. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, UIST ’19, pages 281–292, New York, NY, USA, 2019. Association for Computing Machinery. doi:[10.1145/3332165.3347925](https://doi.org/10.1145/3332165.3347925).
- 18 Yuanming Hu, Luke Anderson, Tzu-Mao Li, Qi Sun, Nathan Carr, Jonathan Ragan-Kelley, and Frédo Durand. DiffTaichi: Differentiable programming for physical simulation, 2020. [arXiv:1910.00935](https://arxiv.org/abs/1910.00935).
- 19 Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. Taichi: A language for high-performance computation on spatially sparse data structures. *ACM Trans. Graph.*, 38(6), November 2019. doi:[10.1145/3355089.3356506](https://doi.org/10.1145/3355089.3356506).
- 20 Michael Innes. Don’t unroll adjoint: Differentiating SSA-form programs, 2019. [arXiv:1810.07951](https://arxiv.org/abs/1810.07951).
- 21 Mike Innes, Alan Edelman, Keno Fischer, Chris Rackauckas, Elliot Saba, Viral B. Shah, and Will Tebbutt. ∂P : A differentiable programming system to bridge machine learning and scientific computing, 2019. [arXiv:1907.07587](https://arxiv.org/abs/1907.07587).
- 22 Wenzel Jakob, Sébastien Speirer, Nicolas Roussel, and Delio Vicini. Dr.Jit: A just-in-time compiler for differentiable rendering. *ACM Trans. Graph.*, 41(4), July 2022. doi:[10.1145/3528223.3530099](https://doi.org/10.1145/3528223.3530099).
- 23 Jerzy Karczmarczuk. Functional differentiation of computer programs. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP ’98, pages 195–203, New York, NY, USA, 1998. Association for Computing Machinery. doi:[10.1145/289423.289442](https://doi.org/10.1145/289423.289442).
- 24 Benjamin Kenwright. Introduction to the WebGPU API. In *ACM SIGGRAPH 2022 Courses*, SIGGRAPH ’22, New York, NY, USA, 2022. Association for Computing Machinery. doi:[10.1145/3532720.3535625](https://doi.org/10.1145/3532720.3535625).
- 25 Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. [arXiv:1412.6980](https://arxiv.org/abs/1412.6980).

- 26 Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004. doi:10.1109/CGO.2004.1281665.
- 27 Dong C. Liu and Jorge Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical programming*, 45(1):503–528, 1989.
- 28 Oleksandr Manzyuk, Barak A. Pearlmutter, Alexey Andreyevich Radul, David R. Rush, and Jeffrey Mark Siskind. Perturbation confusion in forward automatic differentiation of higher-order functions. *Journal of Functional Programming*, 29:e12, 2019. doi:10.1017/S095679681900008X.
- 29 William Moses and Valentin Churavy. Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 12472–12485. Curran Associates, Inc., 2020. URL: <https://proceedings.neurips.cc/paper/2020/file/9332c513ef44b682e9347822c2e457ac-Paper.pdf>.
- 30 William S. Moses, Valentin Churavy, Ludger Paepler, Jan Hückelheim, Sri Hari Krishna Narayanan, Michel Schanen, and Johannes Doerfert. Reverse-mode automatic differentiation and optimization of GPU kernels via Enzyme. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’21, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3458817.3476165.
- 31 William S. Moses, Sri Hari Krishna Narayanan, Ludger Paepler, Valentin Churavy, Michel Schanen, Jan Hückelheim, Johannes Doerfert, and Paul Hovland. Scalable automatic differentiation of multiple parallel paradigms through compiler augmentation. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC ’22. IEEE Press, 2022.
- 32 Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.
- 33 Adam Paszke, Daniel D. Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew J. Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. Getting to the point: Index sets and parallelism-preserving autodiff for pointful array programming. *Proc. ACM Program. Lang.*, 5(ICFP), August 2021. doi:10.1145/3473593.
- 34 Amit Patel. Red Blob Games, 2013. URL: <https://www.redblobgames.com/>.
- 35 Barak A. Pearlmutter and Jeffrey Mark Siskind. Lazy multivariate higher-order forward-mode AD. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’07, pages 155–160, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1190216.1190242.
- 36 Barak A. Pearlmutter and Jeffrey Mark Siskind. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Trans. Program. Lang. Syst.*, 30(2), March 2008. doi:10.1145/1330017.1330018.
- 37 Pyodide contributors and Mozilla. Pyodide, 2019. URL: <https://pyodide.org/>.
- 38 Dan Quinlan and Chunhua Liao. The ROSE source-to-source compiler infrastructure. In *Cetus Users and Compiler Infrastructure Workshop, in conjunction with PACT*, volume 2011, page 1. Citeseer, 2011.
- 39 Alexey Radul, Adam Paszke, Roy Frostig, Matthew J. Johnson, and Dougal Maclaurin. You only linearize once: Tangents transpose to gradients. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. doi:10.1145/3571236.

- 40 Amir Shaikhha, Andrew Fitzgibbon, Simon Peyton Jones, and Dimitrios Vytiniotis. Destination-passing style for efficient memory management. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, FHPC 2017, pages 12–23, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3122948.3122949.
- 41 Xipeng Shen, Guoqiang Zhang, Irene Dea, Samantha Andow, Emilio Arroyo-Fang, Neal Gafter, Johann George, Melissa Grueter, Erik Meijer, Olin Grigsby Shivers, Steffi Stumpos, Alanna Tempest, Christy Warden, and Shannon Yang. Coarsening optimization for differentiable programming. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021. doi:10.1145/3485507.
- 42 Jeffrey Mark Siskind and Barak A Pearlmutter. Nesting forward-mode AD in a functional framework. *Higher-Order and Symbolic Computation*, 21(4):361–376, 2008.
- 43 Jeffrey Mark Siskind and Barak A. Pearlmutter. Divide-and-conquer checkpointing for arbitrary programs with no user annotation. *Optimization Methods and Software*, 33(4-6):1288–1330, 2018. doi:10.1080/10556788.2018.1459621.
- 44 Tom J. Smeding and Matthijs I. L. Vákár. Efficient dual-numbers reverse AD via well-known program transformations. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. doi:10.1145/3571247.
- 45 Tom J. Smeding and Matthijs I. L. Vákár. Efficient CHAD. *Proc. ACM Program. Lang.*, 8(POPL), January 2024. doi:10.1145/3632878.
- 46 Daniel Smilkov, Nikhil Thorat, Yannick Assogba, Charles Nicholson, Nick Kreeger, Ping Yu, Shanqing Cai, Eric Nielsen, David Soegel, Stan Bileschi, Michael Terry, Ann Yuan, Kangyi Zhang, Sandeep Gupta, Sarah Sirajuddin, D Sculley, Rajat Monga, Greg Corrado, Fernanda Viegas, and Martin M Wattenberg. TensorFlow.js: Machine learning for the web and beyond. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 309–321, 2019. URL: https://proceedings.mlsys.org/paper_files/paper/2019/file/acd593d2db87a799a8d3da5a860c028e-Paper.pdf.
- 47 Bobbie Soedirgo. Compile and run LLVM IR in the browser, October 2023. original-date: 2021-02-24T14:29:16Z. URL: <https://github.com/soedirgo/llvm-wasm>.
- 48 Bert Speelpenning. *Compiling Fast Partial Derivatives of Functions Given by Algorithms*. PhD thesis, University of Illinois at Urbana-Champaign, 1980. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2023-02-19. URL: <https://www.proquest.com/dissertations-theses/compiling-fast-partial-derivatives-functions/docview/302969224/se-2>.
- 49 Matthijs Vákár and Tom Smeding. CHAD: Combinatory homomorphic automatic differentiation. *ACM Trans. Program. Lang. Syst.*, 44(3), August 2022. doi:10.1145/3527634.
- 50 Bret Victor. Explorable explanations, 2011. URL: <https://worrydream.com/ExplorableExplanations/>.
- 51 Fei Wang, Daniel Zheng, James Decker, Xilun Wu, Grégory M. Esertel, and Tiark Rompf. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019. doi:10.1145/3341700.
- 52 Guillermo Webster. g9: Automatically interactive graphics, 2016. URL: <https://omrelli.ug/g9/>.
- 53 Yuting Yang, Connelly Barnes, Andrew Adams, and Adam Finkelstein. A δ : Autodiff for discontinuous programs – Applied to shaders. *ACM Trans. Graph.*, 41(4), July 2022. doi:10.1145/3528223.3530125.
- 54 Katherine Ye, Wode Ni, Max Krieger, Dor Ma’ayan, Jenna Wise, Jonathan Aldrich, Joshua Sunshine, and Keenan Crane. Penrose: From mathematical notation to beautiful diagrams. *ACM Trans. Graph.*, 39(4), August 2020. doi:10.1145/3386569.3392375.

Mover Logic: A Concurrent Program Logic for Reduction and Rely-Guarantee Reasoning

Cormac Flanagan 

University of California, Santa Cruz, CA, USA

Stephen N. Freund  

Williams College, Williamstown, MA, USA

Abstract

Rely-guarantee (RG) logic uses thread interference specifications (relies and guarantees) to reason about the correctness of multithreaded software. Unfortunately, RG logic requires each function postcondition to be “stabilized” or specialized to the behavior of other threads, making it difficult to write function specifications that are reusable at multiple call sites.

This paper presents mover logic, which extends RG logic to address this problem via the notion of atomic functions. Atomic functions behave as if they execute serially without interference from concurrent threads, and so they can be assigned more general and reusable specifications that avoid the stabilization requirement of RG logic. Several practical verifiers (Calvin-R, QED, CIVL, Armada, Anchor, etc.) have demonstrated the modularity benefits of atomic function specifications. However, the complexity of these systems and their correctness proofs makes it challenging to understand and extend these systems. Mover logic formalizes the central ideas of reduction in a declarative program logic that provides a foundation for future work in this area.

2012 ACM Subject Classification Theory of computation → Program verification; Theory of computation → Program specifications; Software and its engineering → Concurrent programming languages; Software and its engineering → Formal software verification

Keywords and phrases concurrent program verification, reduction, rely-guarantee reasoning, synchronization

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.16

Related Version *Extended Version:* <https://arxiv.org/abs/2407.08070> [20]

Funding This work was supported by NSF grants 2243636 and 2243637.

1 Introduction

Verifying that a multithreaded software system behaves correctly for all possible inputs and thread interleavings is a critically important problem in computer science. To verify large systems, verification techniques must employ *modular reasoning* in which each function’s implementation is verified with respect to its specification. In a multithreaded system, writing precise and reusable function specifications is a rather difficult challenge, since concurrent threads can observe and change the state of a function call not just in its initial and final states, but also at any intermediate states during the function’s execution. Thus, function specifications must describe not just the function’s precondition and postcondition, but also how the function may influence and be influenced by other concurrent threads. To address this problem, Rely-Guarantee (RG) logic [33] uses function specifications that include:

- a *guarantee* G describing how each step of the function may update shared state, and
- a *rely assumption* R describing the behavior of interleaved steps of other threads. The rely assumption might, for example, specify that interleaved steps preserve a data invariant.

Under RG logic, however, a function’s postcondition must summarize not only the behavior the function itself but also the behavior of interleaved steps of other threads [56]. Consequently, RG function specifications are often specialized to the rely assumption and data invariants of a particular client, limiting reuse of those function specifications in other clients, as we illustrate in Section 2.

Lipton’s theory of reduction [41] provides a promising approach to address this problem. It uses a commuting argument to show that certain functions are *atomic* and behave as if they execute serially (without interleaved steps of other threads). Consequently, atomic functions do not require interleaved rely assumptions, and they can be precisely specified using preconditions and postconditions that are independent of any specific client.

Reduction has been widely adopted in a variety of software validation tools, including dynamic analyses [17, 54, 55, 9], type systems [50, 24, 23, 22], and other tools [6, 61, 62, 60]. Over the past two decades, software verifiers based on reduction (*e.g.*, Calvin-R [25], QED [15], CIVL [30, 38], Armada [42], and Anchor [19]) have demonstrated the utility of atomic function specifications in verifying sophisticated concurrent code. To date, however, reduction-based verifiers have not been based on an underlying program logic, such as RG logic. Instead, their soundness arguments are typically based on monolithic proofs whose complexity inhibits further research. To address this complexity barrier, we present mover logic, which extends RG logic to support atomic function specifications via reduction-based reasoning.

In mover logic, thread interference points are documented with `yield` annotations that have no run-time effect. Mover logic verifies that every sequence of operations between two yield points is reducible and hence amenable to sequential reasoning. In order to verify reducibility, mover logic uses synchronization specifications describing both when each thread can access each shared location and how those accesses commute with concurrent accesses of other threads. In contrast to RG logics that must stabilize all state predicates under the rely assumption, mover logic only needs to stabilize predicates at `yield` points. Atomic functions have no yield points and can thus be specified with traditional pre- and postconditions. Moreover, atomic function specifications need not include a client-specific rely assumption that would limit reuse in other clients that have different rely assumptions or data invariants.

Mover logic is a declarative program logic (similar in style to Hoare Logic and RG Logic) that helps explain and justify many subtle aspects of reduction-based verification, including:

- what code blocks are reducible;
- where `yield` annotations are required;
- which functions are atomic;
- what atomic and non-atomic function specifications mean;
- what reasoning is performed by the verifier;
- why this reasoning is sound; and
- which programs are verifiable or not verifiable, and why.

Mover logic simplifies the soundness proof for any specific verifier, because the proof now must only show that the verifier follows the rules of mover logic.

The presentation of our results proceeds as follows.

- Section 2 illustrates the specification entanglement problem of RG logic and shows how mover logic avoids this problem.
- Section 3 reviews Lipton’s theory of reduction.
- Sections 4 and 5 present an overview of mover logic and additional examples.
- Section 6 formalizes a core multithreaded language.
- Section 7 and 8 present mover logic for this language.
- Sections 9 and 10 discuss related work and summarize our contributions.

For clarity of exposition, our presentation of mover logic targets an idealized multithreaded language that captures the essential complexities of multithreaded function specifications. Extending the logic to more complex languages remains an important topic for future work.

2 Limitations of Rely-Guarantee Logic

We motivate the need for mover logic via the example code in Figure 1 (left). That code consists of:

1. A **counter library** that contains the function `add(n)` that adds `n` to the variable `x` and returns the new value of `x`. The initial value of variable `x` is 0, and it is protected by lock `m`, whose value is either the thread identifier `tid` of the thread holding the lock or 0 if unheld. The lock is initially unheld.
2. A **first client** that creates two threads, and each thread calls `add(2)` multiple times before asserting that `x` is even.

This program verifies under RG logic based on the invariant that `x` is always even. This `even(x)` invariant is a precondition and postcondition for both `add()` and `client()`¹:

```
requires even(x)
ensures even(x)
```

In addition, each step by each thread in the program is guaranteed to preserve this invariant. As a result, each thread can rely on other threads to preserve the invariant:

```
relies even(x)
guarantees even(x)
```

These RG specifications are sufficient to verify that the program does not go wrong by failing the `even(u)` assertion in `client()`, but unfortunately the specification for `add()` is tightly-coupled, or *entangled*, with the `even(x)` data invariant from this particular client. A different client would necessitate revising and re-verifying the counter library, which makes modular verification more challenging and less scalable. For example, the second client in Figure 1 (right) enforces the data invariant `x >= 0`, but it cannot be verified with the `add()` specification entangled with the first client. Others have noted this limitation as well (see, for example, [14, 56]).

2.1 Disentangling RG Specifications: First Attempt

The goal of this paper is to support specifications for library functions like `add()` that are *not* specialized to one particular client. As a first attempt to achieve that goal, the code in Figure 2 (left) uses the following natural postconditions for `add()`, where `\old(x)` and `x` refer to the value of `x` upon function entry and exit, respectively:

```
ensures x == \old(x) + n
ensures \result == x
```

¹ Frame conditions, which specify the locations a function may read or modify, also play a key role in modular function specifications, but we do not consider them in this paper due to lack of space. Extending mover logic with frame conditions, perhaps using ideas from separation logic [47, 49], remains an important topic for future work.

Entangled Rely-Guarantee Specification

Counter Library

```
int x;
lock m;
```

relies even(x)
guarantees even(x)
requires even(x)
requires even(n)
ensures even(x)
ensures even(\result)

```
int add(int n) {
    acquire(m);
    int r = x;
    r = r + n;
    x = r;
    release(m);
    return r;
}
```

Verification Error

Library specification is not general enough to verify Second Client

Entangled Specification
 Library depends on
 Client's even(x) invariant

First Client

```
void main() {
    fork { client(); }
    fork { client(); }
}
```

relies even(x)
guarantees even(x)
requires even(x)
ensures even(x)

```
void client() {
    add(2);
    int u = add(2);
    assert even(u);
}
```

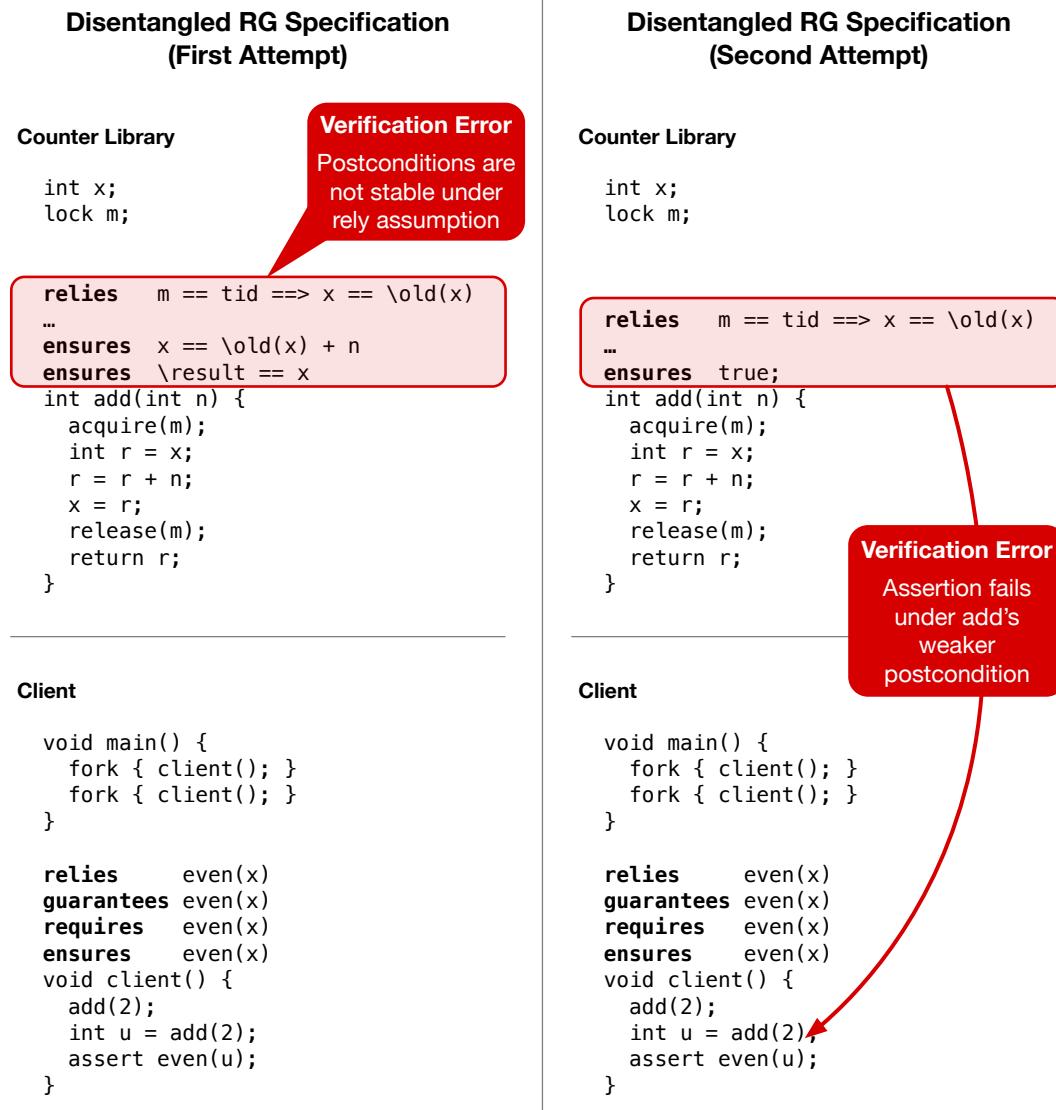
Second Client

```
void main() {
    fork { client(); }
    fork { client(); }
}
```

relies x >= 0
guarantees x >= 0
requires x >= 0
ensures x >= 0

```
void client() {
    add(2);
    int u = add(3);
    assert u >= 0;
}
```

Figure 1 Our idealized running example is an `add(n)` library function that atomically increases shared variable `x` by `n`. (**Left**) A rely-guarantee specification. The client's data invariant `even(x)` becomes entangled in the library specification. (**Right**) A second client that cannot be verified because the specification is insufficiently general.



■ **Figure 2 (Left)** An attempt to disentangle the library specification from the client that does not meet RG stability requirements. **(Right)** Another attempt that meets stability requirements but fails to verify the client.

In addition, if `add()` has no knowledge of its client, it must assume that other client threads could call `add()` with arbitrary arguments at any time, and so the natural rely assumption is that other threads may update `x` whenever the lock `m` is not held by the current thread. That assumption is most easily expressed as its contra-positive (where `tid` is the identifier of the current thread and lock `m` is held by that thread when `m == tid`):

```
relies m == tid ==> x == \old(x)
```

Here, `\old(x)` and `x` refer to the value of `x` before and after an interleaved action of another thread, respectively.

To account for interleaved steps of other threads, a central requirement of RG logic is that all store predicates (*e.g.* preconditions, postconditions, and invariants) used to reason about program behavior must be *stable* under this rely assumption R . This means that interleaved R -steps from other threads must not invalidate those predicates. In the case of `add` in Figure 2 (left), the postcondition `x == \old(x) + n && \result == x` is not stable under the rely assumption R , reflecting that `x` could be concurrently modified after the lock is released but before `add()` returns. Thus, Figure 2 (left) does not verify under RG logic.

2.2 Disentangling RG Specifications: Second Attempt

To ensure stability we must weaken the `add()` function's postcondition to be stable under the rely assumption, as shown in Figure 2 (right). Unfortunately, the resulting stable post condition is simply `true`, which no longer guarantees anything about the value of `x` and is too weak to verify the client.

2.3 Broken Invariants and Bidirectional Entanglement

As a more challenging example, consider the `add()` library variant in Figure 3 (left) that temporarily breaks the `even(x)` invariant while holding the lock. In this case, the invariant holds only when lock `m` is free:

```
m == 0 ==> even(x)
```

Stores at the program points in which the invariant is broken are not intended to be observable by clients. However, the revised RG specifications for `add()` and the client must now be based on this conditional invariant, resulting in two problems. First, the library specification is again specialized to the client's `even(x)` invariant. Second, the library's internal locking discipline leaks into the client's specification, limiting our ability to modify the library code without breaking clients. This example demonstrates that RG reasoning may force us to lose modularity between client and library.

3 Review of Lipton's Theory of Reduction

Our solution to this specification problem employs Lipton's theory of reduction [41], which classifies how steps of one thread commute with concurrent steps of another thread.

- A step is a *right-mover* (R) if it commutes “to the right” of any subsequent step by a different thread, in that performing the steps in the opposite order does not change the final store. A lock acquire is a right-mover because any subsequent step from another thread cannot modify that lock.

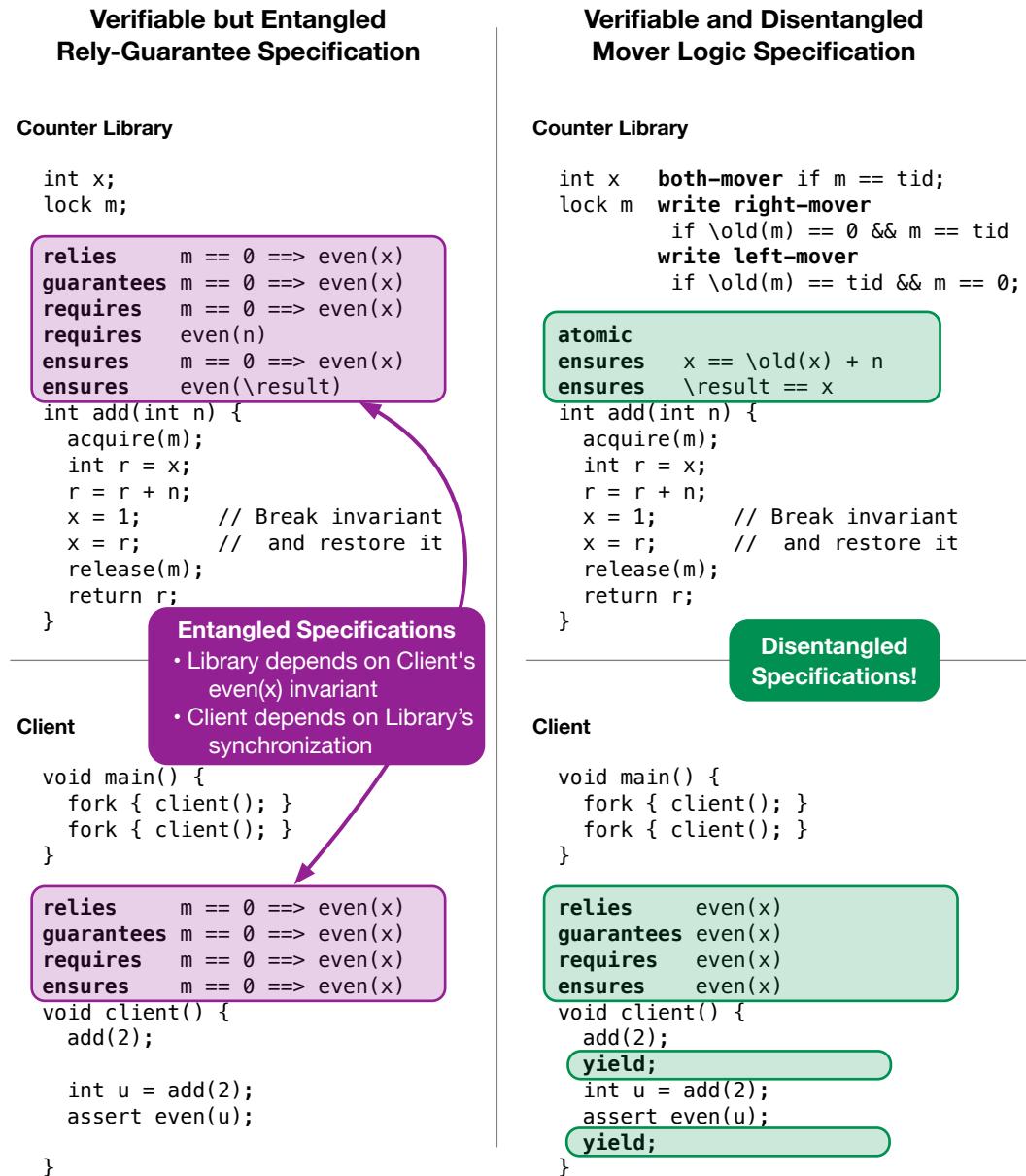


Figure 3 A second version of the counter library with a temporarily broken even(x) invariant. (**Left**) Under RG logic, the library specification is entangled with the client's even(x) invariant and the client specification is entangled with the library's synchronization discipline. (**Right**) Under mover logic, the specifications are cleanly disentangled.

16:8 Mover Logic

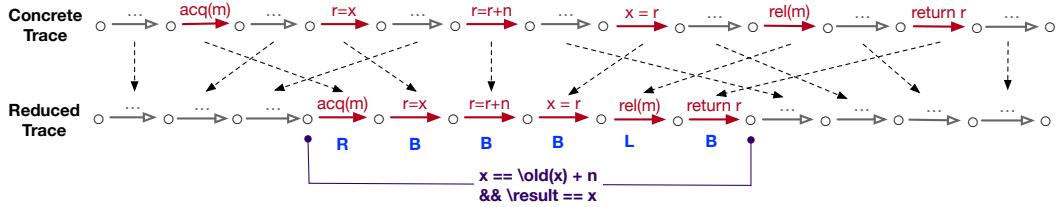


Figure 4 Reduction applied to an execution trace of `add()` from Figure 1.

- Conversely, a step is a *left-mover* (**L**) if it commutes “to the left” of a preceding step of a different thread. A lock release is a left-mover because any preceding step cannot modify that lock.
- A step is a *both-mover* (**B**) if it is both a left- and a right-mover, and it is a *non-mover* (**N**) if neither. A race-free memory access is a both-mover because there are no concurrent, conflicting accesses. An access to a race-prone variable is a non-mover since there may be concurrent writes.

A sequence of steps performed by a particular thread is *reducible* if it consists of (1) zero or more right-movers; (2) at most one non-mover; and (3) zero or more left-movers. That is, a sequence is reducible if the commutativity of the steps match the pattern **R***[**N**]**L***. Any interleaved steps of other threads can be “commuted out” to produce a serial execution.

Figure 4 illustrates this technique for a call to `add()` interleaved with steps of a second thread. In that figure and below, the solid and hollow arrow heads indicate steps from different threads, and arrows labeled “...” represent any number of steps by that thread. The steps of `add()` have the mover behavior **RBBBLB**, matching the reducible pattern **R*[N]L***. Thus we can reason about `add()` as if it executes sequentially and assign it the intuitive postcondition $x == \text{\old}(x) + n \ \&\& \ \text{\result} == x$.

4 Overview of Mover Logic

Mover logic extends RG logic to verify that certain functions are *atomic* and can therefore be assigned more precise (unstabilized) postconditions than under RG logic. Figure 3 (right) shows a mover logic specification for our library/client example. The declaration

```
int x  both-mover if m == tid;
```

means that accesses to `x` are both-movers provided that the current thread holds lock `m`. All other accesses are errors. The declaration for lock `m` specifies that acquires (which change `m` from 0 to the current thread’s identifier `tid`) are right-movers and releases (which change `m` from `tid` back to 0) are left-movers:

```
lock m  write right-mover
    if \old(m) == 0 && m == tid
write left-mover
    if \old(m) == tid && m == 0;
```

These mover specifications are sufficient to verify that `add` is atomic. Consequently, there is no need to apply the rely assumption at each intermediate store inside this atomic function. Instead, sequential reasoning suffices to establish the desired postcondition $x == \text{\old}(x) + n \ \&\& \ \text{\result} == x$.

The `client()` function in Figure 3 (right) is not atomic because steps of other threads could be interleaved between the two calls to `add(2)`. Mover logic uses a `yield` annotation to identify that thread interference may occur at that point, and the store invariants at `yields` must be stable under the rely assumption:

```
relies    even(x);
guarantees even(x);
```

Note that this thread guarantee does not need to summarize individual steps inside the callee `add()`, which would expose the broken invariant. Instead, it summarizes the entire atomic effect of `add()`, which preserves the `even(x)` invariant. With mover logic, the `client()` specification is independent of the internal synchronization discipline inside `add()`.

This library/client example illustrates several benefits of mover logic:

- Verifying that `add()` is atomic enables sequential reasoning inside `add()`.
- We thus avoid applying the rely assumption at each program point inside `add()`.
- As a result, `add()` satisfies the desired postcondition `x == \old(x) + n && \result == x`, which is independent of the client-specific data invariant `even(x)`.
- On the client side, the thread guarantee `even(x)` summarizes the entire behavior of `add()`, rather than the behavior of each individual step.
- Consequently, the client can be verified based on the illusion that `even(x)` always holds, with no loss of soundness.

Thus, mover logic disentangles the library specification from the data invariant of the client while also disentangling the client specification from the library synchronization discipline.

5 Additional Examples

5.1 Spin Lock

To further illustrate the benefits of disentangled specifications, Figure 5 (left) shows our counter library rewritten to employ a user-defined spin lock. The `spin_lock()` code employs a compare-and-set operation (`cas`) to attempt to change the lock 1 from 0 to the current thread's `tid`. The `cas` operation returns true if the update succeeds, and false otherwise. Thus, the function retries until the update is success, at which point the current thread holds the lock. The `spin_unlock()` function releases the lock by setting 1 back to 0.

Mover logic verifies that calls to `spin_lock()` and `spin_unlock()` are atomic right- and left-movers, respectively. That enables us to avoid entangled specifications for the spin lock and counter libraries, and the counter library's `add` specification is *identical* to the earlier implementation. It is still atomic and it guarantees the same post condition.

5.2 Lock-Free Queue

Figure 5 (top right) shows a lock-free single-element queue, where `buf` holds either the single enqueued `int` or `None` if the queue is empty, as indicated by the declared type `Optional[int]`.

The `enqueue(v)` function uses `cas` to switch `buf` from `None` to `v` and is atomic since failing `cas` operations are both-movers. The `dequeue()` function use the action `r ~= buf` to denote an *unstable read* of `buf` that can load any value into the local variable `r` [22]. Unstable reads can be treated as right-movers since they commute past steps by other

16:10 Mover Logic

Verifiable Spin Lock, Counter, Client

Spin Lock Library

```
int l write right-mover
    if \old(l) == 0 && l == tid
    write left-mover
    if \old(l) == tid && l == 0;
```

atomic right-mover
ensures l == tid

```
void spin_lock() {
    while (!cas(l, 0, tid)) {
        skip;
    }
}
```

atomic left-mover
requires l == tid

```
void spin_unlock() {
    l = 0;
}
```

Disentangled Specifications!

Counter Library

```
int x both-mover if l == tid;
```

atomic
ensures x == \old(x) + n
ensures \result == x

```
int add(int n) {
    spin_lock();
    int r = x;
    r = r + n;
    x = 1;           // Break invariant
    x = r;           // and restore it
    spin_unlock();
    return r;
}
```

Disentangled Specifications!

Client

```
void main() {
    fork { client(); }
    fork { client(); }
}
```

relies even(x)
guarantees even(x)
requires even(x)
ensures even(x)

```
void client() {
```

```
    add(2);
```

yield;

```
    int u = add(2);
```

```
    assert even(u);
```

yield;

```
}
```

Verifiable Lock-Free Queue Library

```
Optional[int] buf non-mover;
```

atomic
requires n != None
ensures buf == n

```
void enqueue(int n) {
    while (!cas(buf, None, n)) {
        skip;
    }
}
```

atomic
ensures \result == \old(buf)
ensures buf == None

```
int dequeue() {
    Optional[int] r ~ buf;
    while (r == None) { r ~ buf; }

    while (!cas(buf, r, None)) {
        r ~ buf;
        while (r == None) { r ~ buf; }
    }
    return r;
}
```

Verifiable Lock-Free Stack Library

```
List top non-mover;
```

atomic

ensures head(top) == v
ensures tail(top) == \old(top)

```
void push(int v) {
    List t ~ top;
    List nu = v::t;
    while (!cas(top, t, nu)) {
        t ~ top;
        nu = v::t;
    }
}
```

atomic

ensures head(\old(top)) == \result
ensures tail(\old(top)) == top

```
int pop() {
    List t ~ top;
    while (t == Nil) { t ~ top; }
    List tl = tail(t);

    while (!cas(top, t, tl)) {
        t ~ top;
        while (t == Nil) { t ~ top; }
        tl = tail(t);
    }
    return head(t);
}
```

Figure 5 (Left) A new implementation of the counter library using a user-defined spin lock.
(Top Right) A single-element lock-free queue. **(Bottom Right)** A lock-free stack.

threads.² Consequently, the `dequeue()` function is atomic. All executions of that function consist of unstable reads (right-movers) and failed `cas` operations (both-movers) followed by a successful `cas` (non-mover). These sequences match the reducible pattern $R^*[N]L^*$. Moreover, the final `cas` ensures that `r` is equal to the pre-`cas` value of `buf`, which enables mover logic to establish the desired post-conditions `\result == \old(buf)` and `buf == None`.

5.3 Lock-Free Stack

Figure 5 (bottom right) shows a lock-free stack. This example uses immutable lists, where `Nil` is the empty list, `v::s` adds `v` to the front of the list `s`, and `head(s)` and `tail(s)` extract the first element and the rest of `s`, respectively.³

The `push(v)` function is atomic since it has only one non-mover operation, namely the successful `cas`. The unstable reads, list allocations `v::t`, and failed `cas` operations are both-movers or right-movers. Therefore, we can assign `push(v)` the following intuitive post-condition without needing to stabilize under the rely assumption of a particular caller.

```
ensures head(top) == v
ensures tail(top) == \old(top)
```

The `pop()` function is also atomic due to similar reasoning and satisfies the following post-condition without the need to stabilize it.

```
ensures head(\old(top)) == \result
ensures tail(\old(top)) == top
```

6 Mover Logic Language

We formalize mover logic for the idealized language MML (mover logic language), which we summarize in Figure 6. Section 7.3 below translates our running example into MLL. In MLL, threads manipulate a shared store σ that maps variables to values. Variables include x, y, z , and m . We often use the variable m as a lock, where m is the thread identifier (tid) of the thread holding the lock, or 0 if it is not held.

Thread-local variables r are supported by having each thread access a separate variable r_{tid} for each thread tid . The language includes reads and writes to global and local variables, acquires and releases of locks, local computations, etc. For generality and simplicity, we abstract all of these store-manipulation operations as $actions A \subseteq Tid \times Store \times Store$. Note that an action may depend on the current thread's identifier. We write actions as formulae in which $\text{\old}(x)$ and x to refer to the values of x in the pre-store and post-store, respectively. We write $\langle A \rangle_x$ to denote an action that only changes x :

$$\langle A \rangle_x \stackrel{\text{def}}{=} \{ (tid, \sigma, \sigma') \mid (tid, \sigma, \sigma') \in A \wedge \forall y \in Var. y \neq x \Rightarrow \sigma(y) = \sigma'(y) \}$$

² Unstable reads are a proof technique that trades off our ability to reason about the value stored in `r` for the ability to treat the unstable read as a right-mover. An implementation of unstable read may exhibit any a subset of the allowed behaviors, including simply performing a conventional read.

³ The duplicated code in this example could be removed in a language with richer control structures such as `break` statements.

16:12 Mover Logic

Syntax

(Statements)	$s ::= \text{skip} \mid \text{wrong} \mid A \mid s; s \mid \text{if } C \text{ s else } s$
	$\mid \text{while } C \text{ s } \mid f() \mid \text{yield}$
(Action)	$A \subseteq Tid \times Store \times Store$
(Thread Identifier)	$t, u \in Tid = \{1, 2, \dots\}$
(Conditional Action)	$C ::= A \diamond A$
(Variable Declaration)	$var ::= x \text{ var_spec}$
	$x, y, r, m \in Var$
(Function Declaration)	$fn ::= fn_spec \ f() \ \{ s \}$
	$f \in \text{FunctionName}$
(Declaration Table)	$D ::= \overline{var \mid fn}$
	(D is left implicit in the semantics for brevity)

Semantics

(Store)	$\sigma \in Var \rightarrow Value$
(State)	$\Sigma ::= s_1..s_n \cdot \sigma$
(Evaluation Context)	$E ::= \bullet \mid E; s$

$$s \cdot \sigma \xrightarrow{t} s' \cdot \sigma'$$

[E-SEQ]	$E[\text{skip}; s] \cdot \sigma \xrightarrow{t} E[s] \cdot \sigma$
[E-YIELD]	$E[\text{yield}] \cdot \sigma \xrightarrow{t} E[\text{skip}] \cdot \sigma$
[E-ACTION]	$E[A] \cdot \sigma \xrightarrow{t} E[\text{skip}] \cdot \sigma' \text{ if } (t, \sigma, \sigma') \in A$
[E-IF]	$E[\text{if } (A_1 \diamond A_2) \text{ s}_1 \text{ else } s_2] \cdot \sigma \xrightarrow{t} E[s_i] \cdot \sigma' \text{ if } (t, \sigma, \sigma') \in A_i, \text{ for } i \in 1, 2$
[E-WHILE]	$E[\text{while } C \text{ s}] \cdot \sigma \xrightarrow{t} E[\text{if } C \text{ (s; while } C \text{ s)} \text{ else skip}] \cdot \sigma$
[E-CALL]	$E[f()] \cdot \sigma \xrightarrow{t} E[s] \cdot \sigma \text{ if } fn_spec \ f() \ \{ s \} \in D$

$$\Sigma \rightarrow \Sigma'$$

$$\frac{\begin{array}{c} [\text{E-STATE}] \\ s_t \cdot \sigma \xrightarrow{t} s'_t \cdot \sigma' \end{array}}{s_1..s_t..s_n \cdot \sigma \rightarrow s_1..s'_t..s_n \cdot \sigma'}$$

Figure 6 Mover Logic Language.

We can then express assignments and locking operations as follows. Note that $\text{acquire}(m)$ blocks if the lock is already held, *i.e.* if $\text{old}(m) \neq 0$. We use the notation $expr[x := \text{old}(x)]$ to denote $expr$ with all occurrences of x replaced by $\text{old}(x)$.

$$\begin{aligned} \text{acquire}(m) &\stackrel{\text{def}}{=} \langle \text{old}(m) = 0 \wedge m = tid \rangle_m \\ \text{release}(m) &\stackrel{\text{def}}{=} \langle m = 0 \rangle_m \\ x = expr &\stackrel{\text{def}}{=} \langle x = expr[x := \text{old}(x)] \rangle_x \end{aligned}$$

The unstable read $r_{tid} \sim= x$ from Section 5.3 may store any value⁴ in the local variable r_{tid} :

$$r_{tid} \sim= x \stackrel{\text{def}}{=} \{ (tid, \sigma, \sigma[r_{tid} := v]) \mid v \in Value \}$$

⁴ In a language with types, this definition can be easily adapted to only store type-correct values into r_{tid} .

Mover Logic Language includes `if` and `while` statements that condition execution either on whether a Boolean test is true or on whether a store-manipulating operation, such as `cas`, succeeds. To handle these two cases uniformly, we introduce a *conditional action* $C = A_1 \diamond A_2$ where A_1 is an action capturing a true test or successful operation and A_2 is an action capturing a false test or failed operation. For generality, both cases may modify the store and both may be feasible on some pre-states.

We encode any state predicate $B \subseteq Store$ as the conditional action $\{(tid, \sigma, \sigma) \mid \sigma \in B\} \diamond \{(tid, \sigma, \sigma) \mid \sigma \notin B\}$ that distinguishes the true/false cases but never modifies the store. The following illustrates this encoding for the test $x \geq 0$.

$$x \geq 0 \stackrel{\text{def}}{=} \{(tid, \sigma, \sigma) \mid \sigma(x) \geq 0\} \diamond \{(tid, \sigma, \sigma) \mid \sigma(x) < 0\}$$

As a more interesting example, we encode `cas` as the following conditional action:

$$\text{cas}(x, v, v') \stackrel{\text{def}}{=} \langle \text{old}(x) = v \wedge x = v' \rangle_x \diamond I$$

where the identity action $I = \{(t, \sigma, \sigma) \mid t \in Tid \text{ and } \sigma \in Store\}$. This definition permits `cas` to non-deterministically fail from any pre-state, which enables us to treat failed `cas` operations as both movers [19].

Given $C = A_1 \diamond A_2$, the if statement `if C s1 else s2` may either: 1) evaluate the action A_1 and then s_1 , or 2) evaluate A_2 and then s_2 . The former is the “true” case and the latter is the “false” case, with the desired behavior regardless of whether C encodes a predicate test or a potentially-failing store update. To prevent the if statement from blocking, we require $(A_1 \cup A_2)$ to be total on the state, *i.e.* $\{\sigma \mid (t, \sigma, _) \in (A_1 \cup A_2)\} = State$.

The while statement `while C s` behaves similarly. It iterates as long as C succeeds. We may need to test the negation of a conditional action. The negation of $C = A_1 \diamond A_2$, written $\neg C$, is simply $A_2 \diamond A_1$. The language includes the statement `wrong` to indicate than an error occurred. The statement `assert B` abbreviates `if B skip else wrong`. The goal of mover logic is to verify that programs do not go wrong.

Global variable declarations have the form $x \ var_spec$ and are kept in a global declaration table D . Function declarations have the form $fn_spec f() \{ s \}$ and are also kept in D . Specifications for globals (var_spec) and functions (fn_spec) are described in Sections 7 and 8, respectively. For notational simplicity, D is left as an implicit argument to the evaluation judgments. To keep the core language as simple as possible, we elide formal parameters and return values. Instead, parameters and return values are passed in thread-local variables, as described below in Section 7.3.⁵

In our examples, we include types, curly braces, semicolons, and other standard syntactic forms to aid readability.

An execution state

$$\Sigma = s_1..s_n \cdot \sigma$$

consists of sequence of threads $s_1..s_n$ with a shared store σ . The evaluation relation $\Sigma \rightarrow \Sigma'$ is based on evaluation contexts $E[\dots]$, which identify the next statement to be evaluated. A state $\Sigma = s_1..s_n \cdot \sigma$ is *wrong* if any thread is about to execute `wrong`, *i.e.*, if $s_i = E[\text{wrong}]$. The semantics demonstrates that `yield` annotations have no effect at run time, but they are used in the mover logic described below.

⁵ Extending the language to include function arguments and results is straightforward, but it adds notational complexities that are orthogonal to our core contributions.

7 Mover Logic Effects and Specifications

Mover logic divides the execution of each thread into reducible code sequences that are separated by `yield` statements identifying where thread interference may be observed.

7.1 Effects

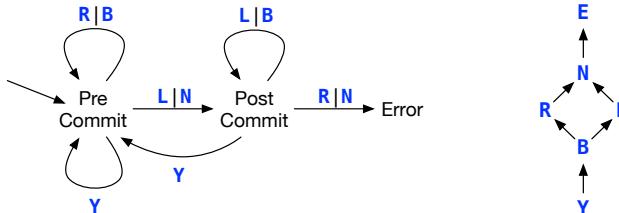
We use a language of effects to reason about reducible code sequences separated by `yields`:

$$e \in \text{Effect} ::= \text{Y} \mid \text{R} \mid \text{L} \mid \text{B} \mid \text{N} \mid \text{E}$$

where

- **Y** is the effect of a `yield` annotation;
- **R** describes right-mover actions;
- **L** describes left-mover actions;
- **B** describes both-mover actions that are both left- and right-movers;
- **N** describes non-mover actions that are neither left- nor right-movers; and
- **E** describes erroneous situations, such as the sequential composition of two non-mover actions without an intervening `yield`, which is not a reducible sequence.

Our strategy for verifying that `yields` correctly separate reducible sequences is based on the DFA [62] shown below (left). The DFA captures reducible sequences $\text{R}^*[\text{N}]\text{L}^*$ separated by yields **Y**, which resets the DFA to the initial “pre-commit” state on the left to start a new reducible sequence. The first left-mover or non-mover in a reducible sequence is often called the *commit* action and moves us from the pre-commit to the post-commit phase.



From this DFA, we derive the ordering $\text{Y} \sqsubseteq \text{B} \sqsubseteq \text{R}, \text{L} \sqsubseteq \text{N} \sqsubseteq \text{E}$, which is also shown above (right). For example, $\text{R} \sqsubseteq \text{N}$, since for any effect sequences α and β , if $\alpha \text{N} \beta$ is accepted by this DFA, the $\alpha \text{R} \beta$ is also accepted. We define a standard join operation \sqcup via this ordering.

We also define sequential composition $e_1; e_2$ and iterative closure e^* , as in [62]. For example, $\text{R}; \text{L} = \text{N}$ since to show $\alpha \text{R} \text{L} \beta$ is accepted by the DFA it is sufficient to show that $\alpha \text{N} \beta$ is accepted. Conversely, $\text{N}; \text{N} = \text{E}$ (error), since $\alpha \text{N} \text{N} \beta$ is never accepted by this DFA.

$e_1; e_2$	$\text{Y} \text{ B} \text{ R} \text{ L} \text{ N} \text{ E}$	e	e^*
Y	Y Y Y L L E	Y	Y
B	Y B R L N E	B	B
R	R R R N N E	R	R
L	Y L E L E E	L	L
N	R N E N E E	N	E
E	E E E E E E	E	E

7.2 Mover Specifications

In mover logic, the verification of a thread tid is performed in the context of a mover specification describing how each program action A starting in the store σ commutes with steps of other threads. Thus, mover specifications M have the type

$$M : Action \times Tid \times Store \rightarrow Effect \setminus \{Y\}$$

For example, if action A is a local computation that only accesses thread-local variables, we would naturally have

$$M(A, tid, \sigma) = B$$

Alternatively, if a global variable x is protected by a lock m , the write action $x = expr$ might have the mover specification

$$M(x = expr, tid, \sigma) = \begin{cases} B & \text{if } \sigma(m) = tid \\ E & \text{otherwise} \end{cases}$$

indicating that the write is a both-mover only if thread tid holds lock m . Otherwise, it is an error. We assume that $expr$ only accesses local variables, and that $M(A, tid, \sigma)$ is never Y since actions do not yield.

We write mover specifications in the source code using the following notation, which is inspired by earlier reduction-based verifiers [30, 19, 21]:

$$\begin{aligned} var_spec &::= var_clause^* \\ var_clause &::= \text{read } e \text{ if } P \mid \text{write } e \text{ if } P \end{aligned}$$

where $P \subseteq Tid \times Store \times Store$ is a two-store predicate describing the pre-store and post-store of the access to x in question. Further, P can depend on the current thread identifier tid . Similar to actions, we write these predicates as formulae in which $\text{old}(y)$ and y to refer to the values of y in the pre-store and post-store, respectively. To determine the mover effect of a variable access, we evaluate the specification clauses in order and take the effect of the first case where the condition P is satisfied. If no clauses apply, the access has the error effect E . More formally, given the specification for a variable x in the source code, we collect the sequence of clauses for reads and writes separately and then create the mover specification M for x as follows:

$$\begin{aligned} \left[\begin{array}{c} \text{read } e_1 \text{ if } P_1 \\ \vdots \\ \text{read } e_n \text{ if } P_n \end{array} \right] &\implies M(r_{tid} = x, tid, \sigma) = \begin{cases} e_1 & \text{if } P_1(tid, \sigma, \sigma) \\ \vdots & \vdots \\ e_n & \text{if } P_n(tid, \sigma, \sigma) \\ E & \text{otherwise} \end{cases} \\ \left[\begin{array}{c} \text{write } e_1 \text{ if } P_1 \\ \vdots \\ \text{write } e_n \text{ if } P_n \end{array} \right] &\implies M(x = expr, tid, \sigma) = \begin{cases} e_1 & \text{if } P_1(tid, \sigma, \sigma[x := \sigma(expr)]) \\ \vdots & \vdots \\ e_n & \text{if } P_n(tid, \sigma, \sigma[x := \sigma(expr)]) \\ E & \text{otherwise} \end{cases} \end{aligned}$$

where r_{tid} is a local variable, $expr$ only accesses thread-local variables, $\sigma(expr)$ is the result of evaluating $expr$ in the store σ , and the cases for M are evaluated in the order listed.

Counter Library	Client
<pre> int x both-mover if m == tid lock m write right-mover if \old(m) == 0 && m == tid write left-mover if \old(m) == tid && m == 0 atomic non-mover requires true ensures x == \old(x) + arg1_{tid} ensures result_{tid} == x add() { R (\old(m) == 0 & m == tid)_m B r_{tid} = x; B r_{tid} = r_{tid} + arg1_{tid}; B x = 1; B x = r_{tid}; L (m == 0)_m; B result_{tid} = r_{tid}; } </pre>	<pre> relies even(x) guarantees even(x) requires even(x) ensures even(x) client() { // even(x) arg1_{tid} = 2; // even(x) && arg1_{tid} == 2 add_{tid}(); // even(x) Y yield; // even(x) arg1_{tid} = 2; // even(x) && arg1_{tid} == 2 add(); // even(x) && even(result_{tid}) u_{tid} = result_{tid}; // even(x) && even(u_{tid}) if even(u_{tid}) skip else wrong; // even(x) Y yield; // even(x) } </pre>

Initial State Σ $(\text{yield}; \text{client}()).(\text{yield}; \text{client}()) \cdot [x := 0, m := 0]$

Figure 7 The example from Figure 3 (right) expressed in Mover Logic Language.

The declaration for a global variable x protected by a lock m is thus written as

```
int x  read  both-mover if m == tid
       write both-mover if m == tid
```

where **both-mover** is syntactic sugar for the effect **B**. (Similarly, we use **left -mover** for **L**, and so on.) In our examples, we abbreviate these identical read and write cases as follows.

```
int x  both-mover if m == tid
```

7.3 Motivating Example, Revisited

Figure 7 expresses our motivating example from Figure 3 (right) in our Mover Logic Language. As mentioned earlier, an access to a thread-local variable r actually accesses a (global) variable r_{tid} that is reserved for use by thread tid . We use thread-local variables to encode function arguments and results. The fork statements are converted into parallel threads in the initial state Σ . We insert a **yield** at the start of each thread in Σ so that the initial state is well-formed under the non-preemptive semantics we introduce in our formal development.

Given this mover specification, mover logic successfully verifies this code. Figure 7 also demonstrates the reasoning carried out by mover logic. The left margin shows the effect of each action and groups those effects into reducible sequences. The **add()** function is a single

reducible sequence, ensuring that we may treat it as atomic. The `client()` function consists of multiple reducible sequences separated by `yields`. We also show invariants demonstrating that `client()` is correct in comments at each program point.⁶

7.4 Additional Mover Specification Examples

Figure 3 (right) showed how mover specifications can capture the synchronization/commuting behavior of lock acquires, lock releases, and lock-protected variable accesses. Our mover specifications are inspired by the ANCHOR verifier, which used mover specifications to capture many synchronization idioms [19, 1].⁷

To illustrate how mover specifications capture more complex synchronization disciplines, suppose the variable `y` is *write-protected* by a lock `m`. That is, lock `m` must be held for all writes to `y` but not necessarily held for reads. Consequently, `y` should be declared volatile if the code is run under a weak memory model. Writes to `y` are non-movers (due to concurrent reads); lock-protected reads are both-movers (because there can be no concurrent writes); and reads without holding the lock are non-movers (due to concurrent writes). Mover specifications capture this synchronization discipline concisely as follows, where the last clause applies only when `m` is not `tid`.

```
int y write non-mover if m == tid
    read both-mover if m == tid
    read non-mover
```

The FASTTRACK dynamic race detector [18, 57] uses a combination of lock-protected and write-protected disciplines to synchronize accesses to some array pointers. We illustrate that discipline for an array pointer `vc`: initially, a `flag` is false and the pointer `vc` is guarded by `lock`; when `flag` becomes true, `vc` becomes write-guarded by `lock`. The mover specification for this discipline is captured by the first four lines in the specification for `vc`:

```
int vc[]      both-mover if !flag && lock == tid
              write non-mover if flag && lock == tid
              read both-mover if flag && lock == tid
              read non-mover if flag
[i]          both-mover if !flag && lock == tid
[i] read    both-mover if flag && (lock == tid || tid == i)
[i] write   both-mover if flag && (lock == tid && tid == i)
```

This idiom enables the algorithm to avoid using a lock to protect all accesses to `vc` but still replace `vc` with a larger array when necessary. The last three lines capture the synchronization discipline for accessing the array entry `vc[i]`, where we use the extended notation “[`i`] `var_clause`” to describe the synchronization cases for actions that access `vc[i]`. That entry is also initially guarded by `lock` when `flag` is false; when `flag` becomes true, the entry `vc[i]` can only be written by thread `i` while holding `lock`, read by any thread while holding the lock, or read by thread `i` without holding the lock. These reads and writes are all both-movers. These rules prevent all conflicting reads and writes, and thus all accesses to `vc[i]` are both-movers under this synchronization discipline.

⁶ In this example, the rely assumption `even(x)` is sufficient for reasoning about `yield` points. In code where live ranges for local variables span yield points, we would add to the rely assumptions the requirement that one thread does not change another thread’s local variables.

⁷ Our syntax for mover specifications is a syntactic variant of the ANCHOR syntax. In essence, our specifications are sequential `var_clauses`, whereas ANCHOR combines these clauses into a single binary decision tree using the syntax `bool_expr ? mover_spec : mover_spec`.

As a final example, consider a concurrent hashtable consisting of a `table` array and a `locks` array, which has length N . The entry `table[i]` is protected by `locks[i % N]`. The `table` reference itself may change when, for example, `table` is replaced with a larger array. To ensure such changes are done without interference, a write to `table` is permitted only when a thread holds *all* locks. In contrast, `table` can be read by a thread holding *any* lock. All such reads and writes are both-movers, as captured by the following mover specification:

```
Entry table[] write both-mover if  $\forall i \in [0, N]. \text{locks}[i] == \text{tid}$ 
      read both-mover if  $\exists i \in [0, N]. \text{locks}[i] == \text{tid}$ 
      [i] both-mover if  $\text{locks}[i \% N] == \text{tid}$ 
```

As illustrated in the previous two examples from the ANCHOR verifier [19], mover specifications can naturally capture synchronization disciplines that vary with the current program state.

A final example comes from the common iterative parallel algorithm pattern in which a synchronization barrier is used to divide the computation into a series of phases. In the even phases, the main thread (with `tid = 0`) updates shared data structures, and in odd phases, worker threads concurrently read data from those structures, as specified below.

```
int z  read both-mover if phase % 2 == 1
      both-mover if phase % 2 == 0 && tid == 0
```

8 Mover Logic

In this section, we show the proof rules for how mover logic handles statements (Section 8.1); function definitions, calls, and specifications (Sections 8.2–8.3); and run-time states (Section 8.4).

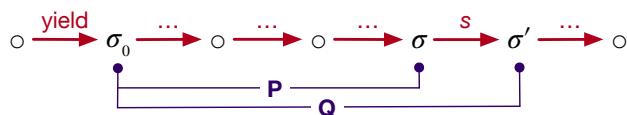
8.1 Mover Logic

Mover logic is defined via the judgments in Figures 8 and 9. The main judgment

$$R; G \vdash s : P \rightarrow Q \cdot e$$

verifies that, when starting from a store satisfying the precondition P , the statement s terminates only in stores satisfying the postcondition Q (*i.e.* partial correctness). In addition, the judgment uses the mover specification M to verify that s consists of reducible sequences separated by `yields`. At each yield point, the rely assumption $R \subseteq Tid \times Store \times Store$ is used to model potential interference from other threads. Conversely, the thread guarantee $G \subseteq Tid \times Store \times Store$ summarizes the behavior of each reducible code sequence between two yield points in s . The effect e summarizes how s commutes with steps of other threads.

In the rules, the precondition P can refer to the value of variable x in the initial store σ_0 of the current reducible code sequence via the notation `\old(x)`. Thus P is a two-store relation $P \subseteq Tid \times Store \times Store$ relating that initial store σ_0 to the pre-store σ for the execution of s . We show that requirement visually in the following trace, where $(tid, \sigma_0, \sigma) \in P$.



One-Store and Two-Store Predicates and Supporting Definitions

$$\begin{aligned}
 S, T &\subseteq Tid \times Store \\
 R, G, P, Q, A &\subseteq Tid \times Store \times Store \\
 Two(S) &\stackrel{\text{def}}{=} \{(t, \sigma, \sigma) \mid (t, \sigma) \in S\} & P; A &\stackrel{\text{def}}{=} \left\{ \begin{array}{l} (t, \sigma, \sigma') \in P \text{ and} \\ (t, \sigma', \sigma'') \in A \end{array} \right\} \\
 Post(P) &\stackrel{\text{def}}{=} \{(t, \sigma) \mid (t, _, \sigma) \in P\} & Yield(P, R) &\stackrel{\text{def}}{=} \left\{ \begin{array}{l} (t, \sigma', \sigma') \in P \text{ and} \\ (t, _, \sigma) \in R^* \end{array} \right\} \\
 I &\stackrel{\text{def}}{=} \{(t, \sigma, \sigma) \mid t \in Tid, \sigma \in Store\}
 \end{aligned}$$

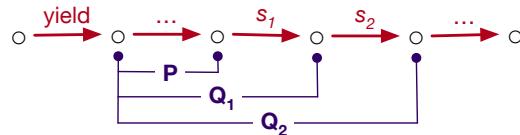
Mover Logic Proof Rules

$\boxed{R; G \vdash s : P \rightarrow Q \cdot e}$	
<p>[M-ACTION]</p> $ \frac{\begin{array}{c} M(A, P) = e \\ e \sqsubseteq \mathbf{L} \Rightarrow A \text{ is total} \end{array}}{R; G \vdash A : P \rightarrow (P; A) \cdot e} $	<p>[M-SEQ]</p> $ \frac{\begin{array}{c} R; G \vdash s_1 : P \rightarrow Q_1 \cdot e_1 \\ R; G \vdash s_2 : Q_1 \rightarrow Q_2 \cdot e_2 \end{array}}{R; G \vdash s_1; s_2 : P \rightarrow Q_2 \cdot (e_1; e_2)} $
<p>[M-IF]</p> $ \frac{\begin{array}{c} R; G \vdash s_1 : P; A_1 \rightarrow Q \cdot e_1 \\ R; G \vdash s_2 : P; A_2 \rightarrow Q \cdot e_2 \\ e = (M(A_1, P); e_1) \sqcup (M(A_2, P); e_2) \end{array}}{R; G \vdash \text{if } (A_1 \diamond A_2) \text{ } s_1 \text{ else } s_2 : P \rightarrow Q \cdot e} $	<p>[M-WHILE]</p> $ \frac{\begin{array}{c} R; G \vdash s : P; A_1 \rightarrow P \cdot e_1 \\ e = (M(A_1, P); e_1)^*; M(A_2, P) \\ e \not\sqsubseteq \mathbf{L} \end{array}}{R; G \vdash \text{while } (A_1 \diamond A_2) \text{ } s : P \rightarrow P; A_2 \cdot e} $
<p>[M-SKIP]</p> $ \frac{}{R; G \vdash \text{skip} : P \rightarrow P \cdot \mathbf{B}} $	<p>[M-WRONG]</p> $ \frac{}{R; G \vdash \text{wrong} : \emptyset \rightarrow \emptyset \cdot \mathbf{B}} $
<p>[M-CONSEQ]</p> $ \frac{\begin{array}{c} P \Rightarrow P_1 \quad R \Rightarrow R_1 \\ Q_1 \Rightarrow Q \quad G_1 \Rightarrow G \quad e_1 \sqsubseteq e \\ R_1; G_1 \vdash s : P_1 \rightarrow Q_1 \cdot e_1 \end{array}}{R; G \vdash s : P \rightarrow Q \cdot e} $	<p>[M-YIELD]</p> $ \frac{\begin{array}{c} P \Rightarrow G \\ Q = Yield(P, R) \end{array}}{R; G \vdash \text{yield} : P \rightarrow Q \cdot \mathbf{Y}} $

Figure 8 Mover logic proof rules and supporting definitions.

The two-store postcondition $Q \subseteq Tid \times Store \times Store$ relates σ_0 to the post-store σ' of s .

Many of the mover logic rules are extensions of Hoare logic incorporating reduction effects. For example, the rule [M-SEQ] states that a sequential composition $(s_1; s_2)$ commutes as $e_1; e_2$, the sequential composition of the effects of its sub-statements, and that the precondition and postcondition are related as follows:



The rule [M-SKIP] indicates that `skip` has no effect, so its precondition and postcondition are identical. The rule [M-WRONG] verifies that `wrong` is never executed via the unsatisfiable precondition \emptyset . That is, this rule rejects any program that may execute `wrong` from any state.

16:20 Mover Logic

Function Specification Syntax

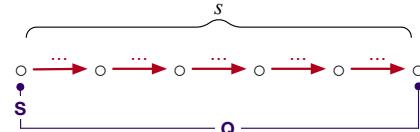
$$fn_spec ::= \begin{array}{l} \text{atomic } e \text{ requires } S \text{ ensures } Q \\ | \quad \text{relies } R \text{ guarantees } G \text{ requires } S \text{ ensures } T \end{array}$$

Proof Rules for Function Definitions and Calls

$\boxed{\vdash fn}$

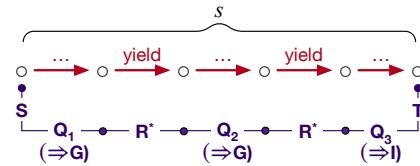
[M-DEF-ATOMIC]

$$\frac{f() \text{ is not (directly or indirectly) recursive} \quad \emptyset; \emptyset \vdash s : Two(S) \rightarrow Q \cdot e}{\vdash \text{atomic } e \text{ requires } S \text{ ensures } Q \quad f() \{ s \}}$$



[M-DEF-NON-ATOMIC]

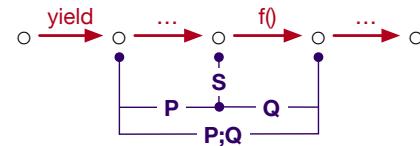
$$\frac{R; G \vdash s : Two(S) \rightarrow Two(T) \cdot R \quad G \neq \emptyset}{\vdash \text{relies } R \text{ guarantees } G \text{ requires } S \text{ ensures } T \quad f() \{ s \}}$$



$\boxed{R; G \vdash s : P \rightarrow Q \cdot e}$

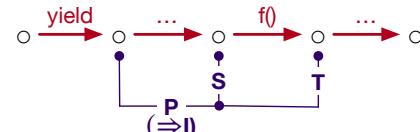
[M-CALL-ATOMIC]

$$\frac{\text{atomic } e \text{ requires } S \text{ ensures } Q \quad f() \{ s \} \in D \quad Post(P) \Rightarrow S}{R; G \vdash f() : P \rightarrow (P; Q) \cdot e}$$



[M-CALL-NON-ATOMIC]

$$\frac{\text{relies } R \text{ guarantees } G \text{ requires } S \text{ ensures } T \quad f() \{ s \} \in D}{R; G \vdash f() : Two(S) \rightarrow Two(T) \cdot R}$$



Verification of States

$\boxed{\vdash \Sigma}$

[M-STATE]

$$\frac{\forall fn \in D. \vdash fn \quad M \text{ is valid} \quad I \Rightarrow G \quad \forall t \in Tid. \left[\begin{array}{l} R; G \vdash s_t : P_t \rightarrow Q_t \cdot e_t \text{ and } e_t \neq E \text{ and } Q_t \Rightarrow G \\ \text{and } s_t \text{ is yielding and } (t, \sigma, \sigma) \in P_t \end{array} \right] \quad \forall t, u \in Tid. t \neq u \Rightarrow (G[tid := t] \Rightarrow R[tid := u])}{\vdash s_1..s_n \cdot \sigma}$$

Figure 9 Mover logic proof rules for function definition, calls, and run-time states.

The rule [M-ACTION] computes the effect of action A from states σ satisfying the current precondition P . That rule uses the function to compute this effect:

$$M(A, P) \stackrel{\text{def}}{=} \bigsqcup_{(t, _, \sigma) \in P} M(A, t, \sigma)$$

(Note that we are overloading M here.) The postcondition of A is then the precondition P sequentially composed with the action A , *i.e.* $P; A$. A key technical requirement of the reduction theorem is that once an atomic block $R^*[\mathbf{N}]L^*$ enters its post-commit (or left-mover part), then it must terminate. It cannot block or diverge [24].⁸ Hence, we require that A is total if it is a left-mover. We place similar restrictions on loops.

The rule [M-IF] requires both the true case $(A_1; s_1)$ and the false case $(A_2; s_2)$ to have the same post-condition Q . The effect e is the maximal effect of executing either A_1 followed by s_1 or A_2 followed by s_2 . The rule [M-WHILE] for `while` $A_1 \diamond A_2\; s$ checks that a successful test followed by the body preserves precondition P , which functions as a loop invariant. The postcondition of the loop is the postcondition of A_2 given the precondition P . The effect of a loop is the iterative closure of the effect of one iteration sequentially composed with the effect of the loop-terminating test A_2 .

Consider the loop in `spin_lock()` in Figure 5. The test `!cas(1, 0, tid)` is the conditional action $I \diamond (\text{\textbackslash old}(1) = 0 \wedge 1 = \text{tid})_1$ and the loop body is `skip`. Since $P; I = P$, rule [M-SKIP] concludes that $R; G \vdash \text{skip} : P \rightarrow P \cdot \mathbf{B}$. Further, $M(I, P) = \mathbf{B}$, because that action accesses no global variables, and the specification for 1 indicates that $M((\text{\textbackslash old}(1) = 0 \wedge 1 = \text{tid})_1, P) = \mathbf{R}$. Thus, $e = (\mathbf{B}; \mathbf{B})^*; \mathbf{R} = \mathbf{R}$. Also, the postcondition $P; A_2$ for the loop simplifies to the expected $P[1 := \text{tid}]$. To ensure the left-mover termination requirement, rule [M-WHILE] requires that $e \not\subseteq \mathbf{L}$. That is, the post-commit part of a reducible sequence cannot contain loops.

The rule [M-YIELD] for `yield` first checks that the thread guarantee G includes all possible behaviors P of the reducible sequence preceding the `yield` via the antecedent $P \Rightarrow G$. The reducible sequence following the `yield` starts with postcondition $Q = \text{Yield}(P, R)$ which incorporates repeated thread interference from other threads via the iterated rely assumption R^* and then resets each `\old(x)` value to be the current value of x at the start of the new reducible sequence.

The rule [M-CONSEQ] extends the consequence rule of RG logic to reduction effects.

8.2 Atomic Functions

Mover logic supports both atomic and non-atomic functions. An atomic function is one whose code body is reducible (*i.e.*, no `yield` statements) and has the following form:

```
atomic e
requires S ensures Q f() { s }
```

(We elide e in the surface syntax when it is \mathbf{N} , as in Figure 3 (right)). The precondition $S \subseteq Tid \times Store$ describes valid initial stores for the function call and must be established by the caller. The post condition $Q \subseteq Tid \times Store \times Store$ describes possible final stores, and it may refer to values of variables on function entry using the `\old(x)` notation. Since s is

⁸ To motivate this requirement consider the program `(x = 1; while (true) skip; yield) || (assert x != 1)`. This program can go wrong because the first thread writes 1 to x . However, the reducible block containing that write never terminates after performing that write, and that write is not included in the thread guarantee G . Thus, we require that once a reducible block commits, it must terminate.

atomic and `yield`-free, we elide the `rely` and `guarantee` components from atomic function specifications. We require atomic functions to be non-recursive to facilitate the “left-mover terminates” requirement mentioned above.

To ensure that the function body s conforms to the function’s specification, rule [M-DEF-ATOMIC] in Figure 9 first converts S into the two-store precondition $\text{Two}(S)$ (in which $\text{old}(x) = x$ for all variables x) and then verifies the function body s with respect to that precondition. We use the guarantee \emptyset to enforce that s is indeed yield-free. (Rule [M-YIELD] will always fail if G is \emptyset , provided that the `yield` is actually reachable, *i.e.* if $P \neq \emptyset$).

The rule [M-CALL-ATOMIC] for a corresponding call to $f()$ retrieves the above specification from the declaration table D and then ensures that the precondition P at the call site implies the callee’s precondition S . That rule uses $\text{Post}(P)$ to first convert P into a one-state predicate. The postcondition $(P; Q)$ combines the call precondition P with the two-store postcondition Q of the callee, as illustrated in the trace to the right of the rule.

8.3 Non-Atomic Functions

Non-atomic function definitions have the following form:

```
requires R guarantees G
requires S ensures T  f() { s }
```

We include thread rely R and guarantee G components in these function specifications since non-atomic function may include `yield` points where thread interference may occur. For simplicity, we require that non-atomic function calls and returns happen at the start of a reducible sequence. Consequently, the precondition $S \subseteq \text{Tid} \times \text{Store}$ and postcondition $T \subseteq \text{Tid} \times \text{Store}$ are both one-store predicates since there is no need to summarize the preceding reducible sequence.

The rule [M-DEF-NON-ATOMIC] checks that the function body s runs from the precondition $\text{Two}(S)$, possibly via multiple reducible sequences separated by `yields`, to terminate after a final `yield` s in a store satisfying T . Those requirements are enforced by using $\text{Two}(T)$ as the postcondition for s . Further, the body s should end in a `yield`, which from the definition of $e_1; e_2$ entails that the effect of s is at most R . At a call site, the rule [M-CALL-NON-ATOMIC] requires that the current reducible sequence is trivial/empty and meets the function’s one-store precondition S by requiring the precondition $\text{Two}(S)$ prior to the call. The rule also converts the function’s one-store postcondition T to the two-store predicate $\text{Two}(T)$.

8.4 Verifying States

We now define the verification judgment $\vdash \Sigma$ to verify program states $\Sigma = s_1..s_n \cdot \sigma$. The rule [M-STATE] for this judgment in Figure 9 ensures that:

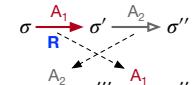
- each thread s_t verifies from a precondition P_t that includes the initial store σ ;
- any pending behavior Q_t at thread termination is published to G ;
- the thread guarantee G is reflexive;
- the guarantee of each thread is contained in the rely assumption of every other thread;
- each function definition in the global table D is verifiable; and
- that all threads start with a `yield` statement (to simplify the correctness proof).

A mover specification M makes claims about how steps of one thread commute with respect to steps of other threads, and mover logic needs to ensure that those claims are correct. Specifically, we define a mover specification to be *valid* if:

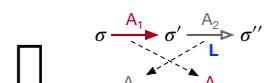
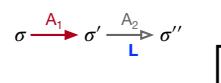
1. Right-moving actions can be moved later in a trace without changing the final store.
 2. Left-moving actions can be moved earlier in a trace without changing the final store.
 3. An action by one thread cannot change the effect of an action in another thread.
 4. An action by one thread cannot cause a left-moving action in another thread to block.
- We formalize these validity requirements as follows:

► **Definition 1** (Validity). M is valid if the following four conditions hold for all threads $t \neq u$ and $A_1, A_2, \sigma, \sigma'$:

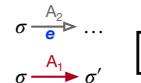
- (1) if $M(A_1, t, \sigma) \sqsubseteq R$ and $(t, \sigma, \sigma') \in A_1$ and
 $M(A_2, u, \sigma') \sqsubseteq L$ and $(u, \sigma', \sigma'') \in A_2$,
then there exists σ''' such that
 $(u, \sigma, \sigma''') \in A_2$ and $(t, \sigma''', \sigma'') \in A_1$.



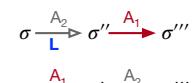
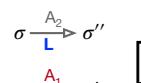
- (2) if $M(A_1, t, \sigma) \sqsubseteq N$ and $(t, \sigma, \sigma') \in A_1$ and
 $M(A_2, u, \sigma') \sqsubseteq L$ and $(u, \sigma', \sigma'') \in A_2$,
then there exists σ''' such that
 $(u, \sigma, \sigma''') \in A_2$ and $(t, \sigma''', \sigma'') \in A_1$.



- (3) if $M(A_1, t, \sigma) \sqsubseteq N$ and $(t, \sigma, \sigma') \in A_1$ and
 $M(A_2, u, \sigma) = e$ for some e ,
then $M(A_2, u, \sigma') = e$.



- (4) if $M(A_1, t, \sigma) \sqsubseteq N$ and $(t, \sigma, \sigma') \in A_1$ and
 $M(A_2, u, \sigma) \sqsubseteq L$ and $(u, \sigma, \sigma'') \in A_2$,
then there exists σ''' such that
 $(u, \sigma', \sigma''') \in A_2$ and $(t, \sigma'', \sigma''') \in A_1$.



8.5 Correctness

The central correctness theorem for mover logic is that verified programs do not go wrong by, for example, failing an assertion.

► **Theorem 2** (Soundness). If $\vdash \Sigma$ then Σ does not go wrong.

The proof appears in full in the extended version of this paper [20]. The basic structure is as follows.

1. We first develop an instrumented semantics that enforces the mover specification M and also that each thread consists of reducible sequences separated by yields.
2. In addition to the usual preemptive scheduler, we also develop a non-preemptive scheduler for the instrumented semantics that context switches only at yields.
3. We show that the instrumented semantics under the preemptive scheduler behaves the same as the standard semantics except that it may go wrong more often.
4. We use a reduction theorem to show that programs exhibit the same behavior under the preemptive and non-preemptive instrumented semantics.
5. Finally, we use a preservation argument [58] to show that verified programs do not go wrong under the non-preemptive instrumented semantics.
6. The steps above then imply that verified programs do not go wrong under the preemptive standard semantics.

9 Related Work

Modular Reasoning

Concurrent software verification introduces a number of scalability challenges that require a synthesis of various notions of modularity or abstraction to address. For example, *procedure-modular* reasoning tackles large code bases by verifying each procedure with respect to a specification of other procedures in the system. Rely-guarantee logic [33] augments procedure-modular reasoning with a notion of *thread-modular* reasoning that accommodates multiple threads by verifying each thread with respect to a specification of other threads in the system. As demonstrated in Section 2, systems like RG logic that support procedure-modular and thread-modular reasoning have great potential, but they are limited by entanglement between library and client specifications.

To address that limitation, mover logic augments procedure-modular and thread-modular reasoning with Lipton’s theory of reduction [41]. This complementary form of “interleaving” modularity limits the number of interleavings that must be considered and enables more precise procedure specifications for atomic functions.

In other work, separation logic combines procedure-modular reasoning with a notion of *heap-modular* reasoning [47, 49], which enables verification of sub-goals while ignoring irrelevant heap objects. Separation logic has been the foundation for a variety of verification tools [3, 32, 44]. Concurrent separation logics including, for example [53, 46, 5, 52], extend those ideas to a concurrent setting. While initially focused on noninterference via disjoint access and read-only sharing, later work [14, 13] supports more tightly-coupled threads.

Much of the work on concurrent separation logic focuses on *resources* (e.g., heap locations) and on ensuring threads access disjoint resources (hence ensuring noninterference). In contrast, mover logic focuses on commuting *actions*.

Concurrent separation logic and mover logic also differ in where thread interference specifications are placed. Concurrent separation logic conveniently merges interference (or resource footprint) specifications into each method’s precondition, thus enabling the logic to capture sophisticated resource usage idioms in a concise and elegant manner. Deny-guarantee reasoning [14] extends concurrent separation logic to focus more on actions rather than resources. In particular, a method’s precondition can include an “action map” specifying what actions the method (and its concurrent threads) may perform. This action map is analogous to our mover specifications. Several projects employ permissions or ownership, similar to separation logic, to reason about which memory locations are available to different threads. These include Viper [43] and VerCors [4]. These systems do not support reduction.

An important topic of future study is how to extend mover logic with a notion of heap modularity, perhaps similar to the core ideas of concurrent separation logic or dynamic frames [2, 51, 35]. This body of work may also provide insight into how to develop a compositional semantics based on mover logic.

Reduction-based Techniques

QED [15] is a program calculus and verification procedure for concurrent programs. It utilizes iterative reduction and abstraction refinement to increase the size of the blocks that can be considered serializable regions (at the abstract level). That approach has been shown to be quite successful for verifying complex concurrent code and has inspired a number of subsequent verification tools described below. Mover logic is a complementary approach in that the combination of RG reasoning and reduction enables direct verification of code

with `yield` points, without the need to create layers of abstractions. As part of that, mover logic supports specifying and reasoning about functions that are not atomic, which is not supported in QED. We also note that QED checks the commutativity properties of an action via a pairwise check with all other actions in the code, whereas mover logic uses the mover specification validity check for that purpose.

Several more recent verification tools utilize the same approach of writing a series of programs related by refinement, abstraction, and reduction. These include the CIVL verifier [30, 38, 40, 36, 37, 39] and the Armada verifier [42]. They are capable of handling sophisticated concurrent code, but do require the programmer to write and maintain multiple versions of the source code. The correctness arguments for these tools have typically been based on monolithic proofs.

Calvin-R [25] developed a number of early ideas related to reduction and thread-modular reasoning. The ANCHOR verifier [19] builds on ideas behind Calvin-R and CIVL to create a verification technique supporting an executable, object-oriented target language, a variety of synchronization primitives, and a new notation for specifying the interference between threads that is the foundation for our mover specifications. While effective at some verification tasks, ANCHOR’s correctness arguments are also challenging to understand and build upon. Further, ANCHOR is inherently limited to small programs because it inlines nested calls during verification, with no mechanism for procedure-modular reasoning. Mover logic may provide a useful foundation for a procedure-modular extension of ANCHOR.

The difficulty in assessing the strengths and weaknesses of the tools mentioned above without a robust underlying logic capturing what they do inspired this work. Mover logic may provide such a foundation, detached from any particular full-scale implementation, that it is accessible, general, and extensible. We hope implementations based on mover logic will follow, as the logic clarifies exactly what conditions must be met in reduction-based verifiers that attempt to integrate modular reasoning in the presence of interference.

Coq-based Techniques

Complementary approaches develop proof frameworks for verifying concurrent programs in Coq [12]. For example, CCAL [28] provides a compositional semantic model for composing and verifying the correctness of multithreaded components. CCAL focuses on only rely-guarantee reasoning [33] and not reduction. CSpec [7] is a Coq library for verifying concurrent systems modeled in Coq [12] using movers and reduction. While highly expressive, particularly because additional proof techniques can be added as additional Coq code, users must write significant Coq code for both specifications and proofs to use such a system. We have focused on a logic more amenable to fully automatic reasoning. Iris [34] uses higher-order separation logic to verify correctness of higher-order imperative programs.

Model Checking

An orthogonal approach to software verification utilizes explicit state, exhaustive model checking. Such approaches have lower programmer overhead than other techniques, but they are non-modular [16, 10, 11]. Specialized techniques, including reduction [29] and partial-order methods [27, 26, 48], have been used to limit state-space explosion while checking concurrent programs. A variety of concurrent software model checkers [8, 59, 45] have demonstrated the potential of these approaches in constrained settings.

10 Summary

Over the last two decades, several promising multithreaded program verifiers have leveraged reduction to verify sophisticated concurrent code including non-blocking algorithms, dynamic data race detectors, and garbage collectors by leveraging precise, reusable specifications for atomic functions. The reasoning used by these verifiers, including the notion of which programs are verifiable, and why the verification process is sound, is unfortunately rather complex. In contrast, Hoare logic [31] provides an accessible foundation for sequential verifiers, and RG logic [33] provides a similar foundation for some multithreaded verifiers.

In developing mover logic, we aim to facilitate future research on reduction-based verification. Mover logic provides a declarative and formal explanation of reduction-based verification, making it easier to understand which programs are verifiable, or not, and why; which functions can be specified as atomic; what atomic and non-atomic function specifications mean; which code blocks are reducible; where yield annotations are required, etc. The correctness proof for a reduction-based verifier need only show that the verifier follows the rules of mover logic, a significant simplification over existing proof techniques.

We hope that mover logic inspires the development of more expressive reduction-based logics and verification tools, potentially supporting features such as objects, data abstraction, dynamic allocation, dynamic thread creation, and precise frame conditions [2, 51, 35].

References

- 1 The Anchor verifier. Accessed: March 30, 2024. URL: <http://www.anchor-verifier.com/>.
- 2 Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Regional logic for local reasoning about global invariants. In *ECOOP*, volume 5142 of *Lecture Notes in Computer Science*, pages 387–411. Springer, 2008.
- 3 Josh Berdine, Cristiano Calcagno, and Peter W. O’Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005.
- 4 Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. The vvercs tool set: Verification of parallel and concurrent software. In *IFM*, volume 10510 of *Lecture Notes in Computer Science*, pages 102–110. Springer, 2017.
- 5 Stephen Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270, 2007.
- 6 Pavol Cerný, Edmund M. Clarke, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, Roopsha Samanta, and Thorsten Tarrach. From non-preemptive to preemptive scheduling using synchronization synthesis. *Formal Methods Syst. Des.*, 50(2-3):97–139, 2017.
- 7 Tej Chajed, M. Frans Kaashoek, Butler W. Lampson, and Nickolai Zeldovich. Verifying concurrent software using movers in CSPEC. In *OSDI*, pages 306–322. USENIX Association, 2018.
- 8 A. T. Chamillard and Lori A. Clarke. Improving the accuracy of petri net-based analysis of concurrent programs. In *ISSTA*, pages 24–38. ACM, 1996.
- 9 Qichang Chen, Liqiang Wang, Zijiang Yang, and Scott D. Stoller. HAVE: detecting atomicity violations via integrated dynamic and static analysis. In *FASE*, volume 5503 of *Lecture Notes in Computer Science*, pages 425–439. Springer, 2009.
- 10 Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.
- 11 Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

- 12 The Coq proof assistant, 2023. URL: <https://coq.inria.fr/>.
- 13 Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hong-seok Yang. Views: compositional reasoning for concurrent programs. In *POPL*, pages 287–300. ACM, 2013.
- 14 Mike Dodds, Xinyu Feng, Matthew J. Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In *ESOP*, volume 5502 of *Lecture Notes in Computer Science*, pages 363–377. Springer, 2009.
- 15 Tayfun Elmas. QED: a proof system based on reduction and abstraction for the static verification of concurrent software. In *ICSE (2)*, pages 507–508. ACM, 2010.
- 16 E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *ICALP*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181. Springer, 1980.
- 17 Cormac Flanagan and Stephen N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, pages 256–267. ACM, 2004.
- 18 Cormac Flanagan and Stephen N. Freund. Fasttrack: efficient and precise dynamic race detection. *Commun. ACM*, 53(11):93–101, 2010.
- 19 Cormac Flanagan and Stephen N. Freund. The Anchor verifier for blocking and non-blocking concurrent software. *Proc. ACM Program. Lang.*, 4(OOPSLA):156:1–156:29, 2020.
- 20 Cormac Flanagan and Stephen N. Freund. Mover logic: A concurrent program logic for reduction and rely-guarantee reasoning (extended version), 2024. [arXiv:2407.08070](https://arxiv.org/abs/2407.08070).
- 21 Cormac Flanagan, Stephen N. Freund, Marina Lifshin, and Shaz Qadeer. Types for atomicity: Static checking and inference for java. *ACM Trans. Program. Lang. Syst.*, 30(4):20:1–20:53, 2008.
- 22 Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. Exploiting purity for atomicity. In *ISSTA*, pages 221–231. ACM, 2004.
- 23 Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *PLDI*, pages 338–349. ACM, 2003.
- 24 Cormac Flanagan and Shaz Qadeer. Types for atomicity. In *TLDI*, pages 1–12. ACM, 2003.
- 25 Stephen N. Freund and Shaz Qadeer. Checking concise specifications for multithreaded software. *J. Object Technol.*, 3(6):81–101, 2004.
- 26 Patrice Godefroid. Model checking for programming languages using verisoft. In *POPL*, pages 174–186. ACM Press, 1997.
- 27 Patrice Godefroid and Pierre Wolper. A partial approach to model checking. In *LICS*, pages 406–415. IEEE Computer Society, 1991.
- 28 Ronghui Gu, Zhong Shao, Jieung Kim, Xiongnan (Newman) Wu, Jérémie Koenig, Vilhelm Sjöberg, Hao Chen, David Costanzo, and Tahina Ramananandro. Certified concurrent abstraction layers. In *PLDI*, pages 646–661. ACM, 2018.
- 29 John Hatcliff, Robby, and Matthew B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *VMCAI*, volume 2937 of *Lecture Notes in Computer Science*, pages 175–190. Springer, 2004.
- 30 Chris Hawblitzel, Erez Petrank, Shaz Qadeer, and Serdar Tasiran. Automated and modular refinement reasoning for concurrent programs. In *CAV (2)*, volume 9207 of *Lecture Notes in Computer Science*, pages 449–465. Springer, 2015.
- 31 C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
- 32 Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and java. In *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011.
- 33 Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

- 34 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018.
- 35 Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2006.
- 36 Bernhard Kragl, Constantin Enea, Thomas A. Henzinger, Suha Orhun Mutluergil, and Shaz Qadeer. Inductive sequentialization of asynchronous programs. In *PLDI*, pages 227–242. ACM, 2020.
- 37 Bernhard Kragl and Shaz Qadeer. Layered concurrent programs. In *CAV (1)*, volume 10981 of *Lecture Notes in Computer Science*, pages 79–102. Springer, 2018.
- 38 Bernhard Kragl and Shaz Qadeer. The civl verifier. In *FMCAD*, pages 143–152. IEEE, 2021.
- 39 Bernhard Kragl, Shaz Qadeer, and Thomas A. Henzinger. Synchronizing the asynchronous. In *CONCUR*, volume 118 of *LIPICS*, pages 21:1–21:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018.
- 40 Bernhard Kragl, Shaz Qadeer, and Thomas A. Henzinger. Refinement for structured concurrent programs. In *CAV (1)*, volume 12224 of *Lecture Notes in Computer Science*, pages 275–298. Springer, 2020.
- 41 Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.
- 42 Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. Armada: low-effort verification of high-performance concurrent programs. In *PLDI*, pages 197–210. ACM, 2020.
- 43 Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *VMCAI*, volume 9583 of *Lecture Notes in Computer Science*, pages 41–62. Springer, 2016.
- 44 Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *Dependable Software Systems Engineering*, volume 50 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 104–125. IOS Press, 2017.
- 45 Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, pages 267–280. USENIX Association, 2008.
- 46 Peter W. O’Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.
- 47 Peter W. O’Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.
- 48 Doron A. Peled. Combining partial order reductions with on-the-fly model-checking. In *CAV*, volume 818 of *Lecture Notes in Computer Science*, pages 377–390. Springer, 1994.
- 49 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.
- 50 Amit Sasturkar, Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. Automated type-based analysis of data races and atomicity. In *PPoPP*, pages 83–94. ACM, 2005.
- 51 Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. An automatic verifier for java-like programs based on dynamic frames. In *FASE*, volume 4961 of *Lecture Notes in Computer Science*, pages 261–275. Springer, 2008.
- 52 Viktor Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, UK, 2008.
- 53 Viktor Vafeiadis and Matthew J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer, 2007.

- 54 Liqiang Wang and Scott D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *PPoPP*, pages 137–146. ACM, 2006.
- 55 Liqiang Wang and Scott D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Software Eng.*, 32(2):93–110, 2006.
- 56 John Wickerson, Mike Dodds, and Matthew J. Parkinson. Explicit stabilisation for modular rely-guarantee reasoning. In *ESOP*, volume 6012 of *Lecture Notes in Computer Science*, pages 610–629. Springer, 2010.
- 57 James R. Wilcox, Cormac Flanagan, and Stephen N. Freund. Verifiedft: a verified, high-performance precise dynamic race detector. In *PPoPP*, pages 354–367. ACM, 2018.
- 58 Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, 1994.
- 59 Eran Yahav. Verifying safety properties of concurrent java programs using 3-valued logic. In *POPL*, pages 27–40. ACM, 2001.
- 60 Jaeheon Yi, Tim Disney, Stephen N. Freund, and Cormac Flanagan. Cooperative types for controlling thread interference in java. In *ISSTA*, pages 232–242. ACM, 2012.
- 61 Jaeheon Yi, Tim Disney, Stephen N. Freund, and Cormac Flanagan. Cooperative types for controlling thread interference in java. *Sci. Comput. Program.*, 112:227–260, 2015.
- 62 Jaeheon Yi and Cormac Flanagan. Effects for cooperable and serializable threads. In *TLDI*, pages 3–14. ACM, 2010.

Fair Join Pattern Matching for Actors

Philipp Haller 

KTH Royal Institute of Technology, Stockholm, Sweden

Ayman Hussein 

Technical University of Denmark, Lyngby, Denmark

Hernán Melgratti 

University of Buenos Aires & Conicet, Argentina

Alceste Scalas 

Technical University of Denmark, Lyngby, Denmark

Emilio Tuosto 

Gran Sasso Science Institute, L'Aquila, Italy

Abstract

Join patterns provide a promising approach to the development of concurrent and distributed message-passing applications. Several variations and implementations have been presented in the literature – but various aspects remain under-explored: in particular, how to specify a suitable notion of message matching, how to implement it correctly and efficiently, and how to systematically evaluate the implementation performance.

In this work we focus on actor-based programming, and study the application of join patterns with conditional guards (i.e., the most expressive and challenging version of join patterns in literature). We formalise a novel specification of *fair and deterministic join pattern matching*, ensuring that older messages are always consumed if they can be matched. We present a *stateful, tree-based join pattern matching algorithm* and prove that it correctly implements our fair and deterministic matching specification. We present a novel Scala 3 actor library (called `JoinActors`) that implements our join pattern formalisation, leveraging macros to provide an intuitive API. Finally, we evaluate the performance of our implementation, by introducing a systematic benchmarking approach that takes into account the nuances of join pattern matching (in particular, its sensitivity to input traffic and complexity of patterns and guards).

2012 ACM Subject Classification Software and its engineering → Formal language definitions; Software and its engineering → Domain specific languages; Software and its engineering → Concurrent programming languages; Software and its engineering → Distributed programming languages; Theory of computation → Process calculi

Keywords and phrases Concurrency, join patterns, join calculus, actor model

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.17

Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact):*

<https://doi.org/10.4230/DARTS.10.2.8>

Funding *Ayman Hussein:* Research supported by the Horizon Europe grant 101093006 (TaRDIS). *Alceste Scalas:* Research partly supported by the Horizon Europe grant 101093006 (TaRDIS).

Emilio Tuosto: Research partly supported by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant agreement No 778233, the PRIN PNRR project DeLICE (P20223T2MF), “by the MUR dipartimento di eccellenza”, and by PNRR MUR project VITALITY (ECS00000041), Spoke 2 ASTRA – Advanced Space Technologies and Research Alliance.

Acknowledgements This work was inspired by the group discussion on “Join patterns / synchronisation – the next generation” [4, page 54] at the Dagstuhl Seminar 21372; we thank the organisers of the meeting and Schloss Dagstuhl – Leibniz Center for Informatics for making this work possible.

 © Philipp Haller, Ayman Hussein, Hernán Melgratti, Alceste Scalas, and Emilio Tuosto;
licensed under Creative Commons License CC-BY 4.0
38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 17; pp. 17:1–17:28



17.2 Fair Join Pattern Matching for Actors

We thank Omar Inverso for the technical support he provided for our experimental evaluation, Roland Kuhn for fruitful discussions on the shop floor use case, António Ravara for some useful suggestions, and Antoine Sébert for an implementation of join patterns using Scala 3 macros [25]. We thank the anonymous reviewers for their comments and suggestions.

1 Introduction

Programming concurrent and distributed message-passing applications is difficult, especially in scenarios where multiple concurrent processes need to synchronise and exchange data when complex conditions are satisfied. The join calculus [8] introduced *join patterns*, an intriguing construct for concurrent programming that can help address these scenarios. A *join pattern with conditional guard* is reminiscent of a clause in a typical pattern matching construct: it has the form “ $J \text{ if } \gamma \triangleright P$ ” – where J is a *message pattern* describing a combination of incoming messages and binding zero or more variables, and γ is a *guard*, i.e., a boolean expression that may use the variables bound in J . A program using join patterns can wait until a desired combination of messages arrives (in any order); when some of the messages are matched by the message pattern J and their payloads satisfy the guard γ , the process P is executed. We now illustrate programming with join patterns with an example emerging from an industrial case study where a monitoring program handles a variety of messages emitted by machines and devices deployed on a factory shop floor. (To illustrate our proposal, we only show a representative sample of the the actual monitoring application.)

```
1 def monitor() = Actor[Event, Unit] {
2     receive { (self: ActorRef[Event]) => {
3         case ( Fault(_, fid1, _, ts1),
4               Fix(_, fid2, ts2) ) if fid1 == fid2 =>
5             updateMaintenanceStats(ts1, ts2)
6             Continue
7
8         case ( Fault(mid, fid1, descr, ts1),
9               Fault(_, fid2, _, ts2),
10            Fix(_, fid3, ts3) ) if fid2 == fid3 && ts2 > ts1 + TEN_MIN =>
11             updateMaintenanceStats(ts2, ts3)
12             log(s"Fault ${fid1} ignored for ${(ts2 - ts1) / ONE_MIN} minutes")
13             self ! DelayedFault(mid, fid1, descr, ts1) // For later processing
14             Continue
15
16         case ( DelayedFault(_, fid1, _, ts1),
17                 Fix(_, fid2, ts2) ) if fid1 == fid2 =>
18             updateMaintenanceStats(ts1, ts2)
19             Continue
20
21         case Shutdown() => Stop
22     }
23 }
```

Listing 1 Simplified factory shop floor maintenance monitor, written using our Scala 3 library `JoinActors` (presented in Section 4).

Example: Monitoring a Factory Shop Floor. The Scala 3 Listing 1 is structured as an actor [1] that uses our join patterns library `JoinActors` (as introduced in Section 4). Its coding style is reminiscent of popular libraries like Akka and Pekko.¹ The constructor `Actor[Event, Unit]` (line 1) means that the actor’s mailbox receives messages of type `Event` (which has various subtypes), and whenever the actor stops running, it yields a `Unit` value. The “`receive { ... }`” block (lines 2–22) executes whenever messages are received, binding `“self”` to a reference to the monitor actor itself (usable to send messages to its mailbox).

The monitor actor in Listing 1 is used in a scenario where machines on the factory shop floor may occasionally require human intervention, so they may emit messages like `Fault(3, 42, "Motion sensor error", 10:31)` carrying information such as the machine and fault identifiers as well as a description and timestamp. When such an event occurs, a technician is expected to reach the machine and report that the fault is being fixed, by using a handheld device to emit a message like `Fix(35, 42, 10:33)` (carrying the worker id, fault id being fixed, and timestamp).

The key difference between the actor depicted in Listing 1 and a “standard” actor in libraries like Akka/Pekko lies in their message processing mechanisms. While the latter can only react to *individual* messages arriving in its mailbox, the actor in Listing 1 reacts whenever a *combination* of messages in its mailbox matches one of the *join patterns with guards* specified within its “`receive { ... }`” block.

- The case on lines 3–4 is triggered when the monitor detects in its mailbox both a `Fault` and a `Fix` message referring to the same fault (guard “`fid1 == fid2`”). In this case, the messages are removed from the mailbox, the monitor updates certain maintenance statistics (line 5), and then resumes execution by returning `Continue` (line 6).
- The case on lines 8–10 activates when the monitor sees *two* `Fault` message and a `Fix` message that handles the most recent fault, with the older fault being emitted more than 10 minutes earlier (guard “`fid2 == fid3 && ts2 > ts1 + TEN_MIN`”). In this case, the monitor also logs a warning and resends the unhandled fault to its own mailbox (as a `DelayedFault`) for later processing (lines 12–13);
- The case on lines 16–17 is similar to the first case above, except that it consumes the `DelayedFaults` emitted by the second case;
- The case on line 21 reacts to a `Shutdown` message by `Stopping` the monitor.

Notice that the join pattern matching cases do not depend on the order of messages in the mailbox: for instance, the first and second cases in Listing 1 (lines 3–4, 8–10) can be triggered even if, due to network delays or temporary partitions, the `Fix` message reaches the monitor mailbox *before* the corresponding `Fault`.²

The monitor in Listing 1 has a declarative and rather intuitive flavour – but writing it without a library (like ours) supporting join patterns is much harder. E.g., to just implement the first and second case (lines 3–4, 8–10), a programmer writing a “regular” Akka/Pekko actor would need to write code for processing one incoming message at a time, remembering how many `Faults` and/or `Fixes` it has seen thus far, and checking whether any combination creates a match with the newly-arrived message, and satisfies the guards; this handcrafted pattern matching logic should not “forget” any message combination, and should also support messages arriving out-of-order. As the number and complexity of message patterns increases, the handcrafted pattern matching code can become complicated, bug-prone, and inefficient.

¹ <https://akka.io/>, <https://pekko.apache.org/>

² The program in Listing 1 only assumes that each device has an accurate-enough clock, so message timestamps can be compared (with some tolerance) to determine which event happened first.

Open Problems. Although promising, join patterns are still subject of research and their adoption has yet to become “mainstream” in programming. In this work we tackle three aspects that, we believe, have been under-explored thus far.

1. *Formalising how a join pattern matching construct should select messages when multiple options are available.* Existing work (both theoretical and implementation-oriented, discussed in Section 2) leaves the message selection unspecified (i.e., allowing for non-determinism in the matching semantics), or follows a “first matching pattern wins” approach – which may cause older messages to be “forgotten” in the mailbox to the advantage of newer messages (we will discuss this in Section 3). This may yield “unfairness” towards the messages in the mailbox: a message in the box is perpetually neglected when “newer” messages are used in the matching.³
2. *Implementing join patterns with guards in a correct and efficient way.* Most existing implementations address message patterns without guards [9, 2, 23, 27, 16]. However, supporting guards is much harder: finding a combination of messages in a mailbox that satisfies a guard may require computing up to a factorial number of message combinations, and in order to reduce such computations, it becomes necessary to maintain the state of partial matches. Other authors have considered this issue [12, 20, 22] – but unlike us, they have either not provided a specific notion of matching nor demonstrated that their optimisation approaches (if any) *correctly* adhere to a desired matching specification (see problem 1 above). In fact, the papers [12, 20] do not define a notion of “preferred matching” while such a notion is mentioned in the doctoral thesis [21] without a formal definition nor proof of the properties of their algorithm.
3. *Systematically evaluating join pattern matching performance.* The performance of join pattern matching is highly dependant on the input message traffic and on the complexity of patterns and guards – but these aspects have not been systematically explored, and there is no standardised benchmarking suite for join pattern implementations (akin to Savina [13] for actor implementations). For instance, the measurements in [27] focus on classic synchronisation problems, with simple patterns, and without guards.

Contributions and Structure of the Paper. We address the aforementioned challenges by presenting a novel formalisation and implementation of join pattern matching with guards. After the background and related work (Section 2), we introduce our contributions.

- In Section 3, we present a formal specification of *fair and deterministic join pattern matching* guaranteeing that oldest messages are always consumed if they can be used (Defs. 3.8 and 3.10). We also introduce a *stateful tree-based matching algorithm* (Defs. 3.20 and 3.23), and we prove that it respects the formal specification of fair matching (Theorem 3.25).
- In Section 4 we present **JoinActors**, our Scala 3 library for actors with join patterns, including both a “brute-force” and a stateful tree-based implementation of our deterministic fair matching semantics. **JoinActors** uses macros to provide an intuitive API. **JoinActors** is the companion artifact of this paper.
- In Section 5 we evaluate the relative performance of the matching algorithms implemented in **JoinActors**, including (in Section 5.6) a comparison with an alternative implementation of our fair matching policy that uses the RETE algorithm [6]. Our evaluation explores variations of the input traffic and the complexity of join patterns and guards: we see this

³ This can be considered a form of *fairness of instruction* according to the terminology in [11].

as a step towards a standardised and systematic benchmarking approach for future join pattern implementations. Overall, our experiments show that the performance of our implementation of the stateful tree-based matching is suitable for applications like floor shop monitoring (described above) or smart house automation (described in Section 5.3).

We conclude and discuss our future work in Section 6 – including alternative matching policies. In this work we have chosen to formalise a “oldest messages first” matching policy because it fits many scenarios – in particular, our factory shop floor monitor (where, as in many application domains, a “first-arrived-first-served policy” is required, and non-deterministic matching would be inadequate), and the examples we could find in literature.

2 Background and Related Work

The Join calculus [8], emerging in the late 1990s as a variant of the asynchronous pi-calculus, aimed at enhancing the implementability of process calculi by introducing disciplined rules regarding locality and scoping. Its distinctive feature is the integration of restriction, recursion, and synchronization into a single language primitive: the *join definition*. A join definition comprises a list of *reaction rules* of the form $J \triangleright P$, where J is the *join pattern* and P is the process associated with the rule. Essentially, a join pattern specifies the message pattern necessary to activate the process P . For instance, in the construct $c_1(x) \wedge c_2(y) \triangleright P$, we have that c_1 and c_2 represent communication channels, and the process P is activated when messages are present in both channels. Hence, if messages like $c_1(m_1)$ and $c_2(m_2)$ are detected, they are consumed, and the process P is executed by substituting the variables x and y with the corresponding values m_1 and m_2 (i.e., P is executed as $P\{m_1, m_2/x, y\}$). A reaction rule can be seen as an evolution of a function definition in a concurrent message-passing setting: a function activates its body upon invocation from another function, through variable substitution – whereas a reaction rule $J \triangleright P$ activates P only when the join pattern J is “invoked” by one or more concurrent processes that send the required input messages; when this happens, P is executed through variable substitution.

Multiple reaction rules can be combined, such as: $c_1(x) \wedge c_2(y) \triangleright P_1 + c_1(z) \wedge c_3(w) \triangleright P_2$. When multiple rules share channels (as c_1 in the previous example), there may be conflicting synchronisations, as rules contend for messages. For example, if channels c_1 , c_2 , and c_3 in the previous example have a message available, either P_1 or P_2 can be activated. Significantly, all conflicting synchronisations are defined within the same combination of reaction rules: consequently, all consumers of messages within a channel are locally introduced by a definition, eliminating the need for global consensus in synchronisation.

Since its inception, the join calculus has inspired implementations in various programming languages [9, 7, 5, 14, 2, 23, 24, 27]. Early implementation approaches [5] were centered around *matching automata*, where join definitions are compiled into deterministic automata. In this cases, state corresponds to the state of message queues and transitions to the arrival of messages. Although the foundational principles of this approach have since been adopted in other implementations [2, 23, 24], newer methods have evolved to avoid the explicit construction of automata. Initially, most implementations relied on coarse-grained synchronization to guarantee the atomic consumption of messages. However, this strategy has been refined [27] by employing fine-grained concurrency for enhanced scalability. This involves the utilization of lock-free data structures and minimizing message enqueueing whenever possible. Subsequent optimizations have further been explored in implementations incorporating session types [10] to prune the size of matching automata.

The initial implementations adhere to the original join pattern model and not support pattern matching on consumed values. However, subsequent implementations expanded matching capabilities. This line of work has been started in [19], where join patterns were enriched with matching on constant values. This approach has been extended in [16] to incorporate pattern matching on algebraic data structures. An example of this extended approach involves message patterns such as $\text{pop}(e) \wedge \text{stack}(x :: xs)$ where the pattern associated with the channel *stack* expects a message containing a non-empty list. In such scenarios, efficient pattern satisfaction can be achieved by translating these extended join patterns into equivalent programs. These programs utilize conventional join patterns in their definitions while incorporating ML-style pattern matching in the processes executed after a join pattern match. Also, [16] showed that *linear* message patterns (i.e., where each bound variable occurs once) without guards can be implemented efficiently by checking bit flags.

The implementation of more expressive forms of pattern matching have been studied in [12, 20, 22]. These works are conceptually more similar to ours: unlike [16], these works support join patterns that include *conditional guards*, i.e., their reaction rules may look like $c_1(x) \wedge c_2(y) \text{ if } x < y \triangleright P$ (resembling pattern matching guards in Erlang or Scala); moreover, these works adapt join patterns to an actor-based setting: in the example above, P is activated when the mailbox of the running actor contains, e.g., the messages $c_1(1)$ and $c_2(2)$ (in any order, and possibly among other messages). The introduction of conditional guards significantly improves the usability of join patterns, but also significantly complicates the implementation of join pattern matching; these works adopt different approaches for finding and selecting a match among incoming messages. Both [12] and [20] adopt a “first-match” approach [26], i.e., given a combination of reaction rules, they select the first one that successfully matches the messages in the mailbox; to find that match, [12] adopts a “stateless brute force” approach (i.e., when the mailbox contains a set of messages that might potentially be matched by a join pattern, it tries all message combinations), while [20] maintains a state containing a cache of partial matches, to reduce unnecessary computations. Also, [22] reportedly adopts a variant of the RETE algorithm [6] to maintain a cache of partial matches (as a *discrimination network*) – but its implementation is not publicly available.

In the join calculus, join definitions have non-deterministic matching policies: when multiple message combinations or patterns are enabled (as we will show in Example 3.5), one option is chosen non-deterministically. Correspondingly, existing work and implementations based on join calculus leave matching policies unspecified, or pick the first pattern that completes a match. However, in scenarios where the message selection policy is critical (as in our factory automation example in Section 1, where earlier events must be handled first), the programmer has to encode the selection logic and maintain complex states to achieve the desired outcome. This paper addresses the issue by formalising *fair and deterministic* join pattern matching, inspired by matching mechanisms in functional languages. Drawing from our real-world factory automation use case, we propose an approach that ensures fair message consumption based on messages “age.” While other application scenarios may require different resolution policies, we argue that such policies should be enforced by library mechanisms. (We discuss some alternative policies in Section 6.) In contrast to prior work, we emphasise the formalisation of properties guaranteed by the matching mechanism: we introduce a formal specification of fair matching and prove that a stateful algorithm effectively implements that specification. We also contribute a comprehensive evaluation, as a first step toward a standard benchmark suite for join pattern implementations.

An interesting effect-handlers-based language is formalised in [3] to program different styles of matching across different message streams. Besides a common connection to the join calculus, a key design difference with our work is that we focus on matching messages within

an actor mailbox. The resulting features and applications are very different (e.g. a message in a mailbox may be matched at a later time, after other messages from the same emitter – which is not possible in a stream-based framework. Also, the work [3] does not address a notion of fair matching among juxtaposed “joins over asynchronously arriving events” that compete over the same input messages: in their modelling, no event binding takes precedence over the other, all iterations proceed independently and concurrently [3, page 67:14–15].

3 Formalisation

In this section we present the formalisation and properties of our approach to join pattern matching. In Section 3.1 we formalise the necessary notation, and in Section 3.2 we present a specification of *deterministic and fair matching*, covering both individual join patterns (Def. 3.8) and definitions (Def. 3.10). Then, in Section 3.3 we present a stateful matching algorithm that implements our fair matching specification (Theorem 3.25) while avoiding unnecessary computations.

3.1 Syntax

To abstractly represent messages, we assume a set \mathbb{C} of *constructors* equipped with a map arity assigning a natural number to each constructor; then, $\text{arity}(c) \geq 0$ is the *arity* of $c \in \mathbb{C}$ (c is a constant symbol if $\text{arity}(c) = 0$). A *message* is either a constructor of arity 0 or a term of the form $c(m_1, \dots, m_n)$ where $c \in \mathbb{C}$, $\text{arity}(c) = n > 0$, and m_i is a message (for $i \in 1..n$). For instance, for the example in Section 1, `Fault(3, 42, "Motion sensor error", 10:31)` is a message, with `Fault` being a constructor of arity 4 (numbers, strings, and timestamps are constant symbols represented in the usual way for readability). Through the paper we use *boolean guards* as pure expressions denoted with γ , using the syntax of boolean expressions of Scala; we also use *mailboxes* denoted as \mathcal{M} , as sequences of messages $m_1 \cdot \dots \cdot m_n$. We will also denote *variables* with the symbols y, w, z, \dots

Intuitively, a join pattern is a combination of “messages with variables” that binds the variables occurring therein. Multiple alternative join patterns can be composed in a *join definition*. In Def. 3.1 we formalise join patterns equipped with guards, and join definitions.

► **Definition 3.1** (Join patterns and join definitions). *The syntax of join patterns Π and join definitions D is given by the following grammar:*

$$\begin{aligned} \Pi &::= J \text{ if } \gamma & \text{where } J ::= \mu \mid \mu \wedge J \quad \text{and} \quad \mu ::= m \mid x \mid c(\mu_1, \dots, \mu_{\text{arity}(c)}) \\ D &::= \Pi \mid \Pi + D \end{aligned}$$

We postulate that J if γ must be well-formed, namely: (i) linear, i.e., no variable in J occurs more than once, and (ii) closed, i.e., each variable occurring in the guard γ also occurs in J . We will often simply write J as shorthand for J if `true`.

Assumption (i) in Def. 3.1 is quite standard: e.g., Scala and F# require linear use of pattern matching variables, and non-linear use can be simulated using guards (see Example 3.2 below). Assumption (ii) does not limit our results: in fact, if a guard contains variables bound elsewhere in the surrounding program, then all such variables would be substituted by values (thus “closing” the join pattern) *before* any match is attempted.

► **Example 3.2** (Well-formedness of join patterns). The join patterns shown in Example 3.4 (and also in Listing 1) are well-formed, since they are both linear and closed, whereas

$$\text{Fault}(mid_1, fid, descr_1, ts_1) \wedge \text{Fix}(wid_2, fid, ts_2)$$

is not-well formed, since the double occurrence of variable fid violates linearity. Intuitively, repeating fid can be a convenient way to state that the same fault id fid must appear in both messages. This is not supported by our formalisation – but the same effect can be obtained by linearising the join pattern: it is sufficient to rename the variables into fid_1 and fid_2 and introduce a guard $\text{fid}_1 = \text{fid}_2$, obtaining the first pattern shown in Example 3.4 below. \square

We write $\{m_1, \dots, m_n/x_1, \dots, x_n\}$ for a *substitution*, that is a map that replaces each variable x_i for message m_i . A substitution σ can be applied to join patterns and guards; for instance, the application of the substitution $\sigma = \{42/x\}$ to the join pattern $J = \text{Message}(x)$, written $J\sigma$, yields $\text{Message}(42)$. Similarly $(\text{isOdd}(x))\sigma = \text{isOdd}(42)$.

► Remark 3.3. A typical join pattern rule has the form $J \text{ if } \gamma \triangleright P$. Following the formalisation of ML-style pattern matching in [17], we omit the continuation process P to focus on the matching semantics. Adding continuations P to our formalisation is routine: it would be enough to apply substitutions σ produced by a match to the omitted process, as $P\sigma$.

Intuitively, a mailbox $M = m_1 \cdot \dots \cdot m_n$ yields a match for the join pattern $\Pi = \mu_1 \wedge \dots \wedge \mu_m \text{ if } \gamma$ in D if there is a substitution σ replacing all the variables in Π with some of the messages in the mailbox, such that $\gamma\sigma$ holds `true`; each message in M can be used at most once. A *variable* x matches any message, whereas a message constructor pattern like $\text{Fault}(x, 42, y, w)$ can only match a message built with a corresponding constructor, like $\text{Fault}(3, 42, \text{"Sensor error"}, 10:31)$ with the substitution $\{3, \text{"Sensor error"}, 123456/x,y,w\}$. (For the precise matching semantics, see in Section 3.2.) Observe that when all variables in a join pattern Π are substituted we obtain one or more \wedge -separated messages; likewise, when all variables in a guard γ are substituted, we can evaluate the boolean expression (e.g., a predicate like $\gamma\sigma = \text{isOdd}(42)$ might evaluate to `false`).

A join definition $D = \Pi_1 + \dots + \Pi_k$ specifies a pattern matching operation among one of the join patterns with guards $\Pi_1 \dots \Pi_k$. This formal notation abstracts the construct `receive { ... }` shown in Listing 1: each `case` in the `receive { ... }` is a join pattern in D .

► Example 3.4 (Syntax of join definitions). Assuming $\text{Fault}, \text{Fix} \in \mathbb{C}$ and adopting the syntax in Def. 3.1 and of boolean Scala expressions, the first two join patterns in Listing 1 are:

$$\begin{aligned} & \text{Fault}(mid_1, fid_1, descr_1, ts_1) \wedge \text{Fix}(wid_2, fid_2, ts_2) && \text{if } fid_1 = fid_2 \\ + \quad & \text{Fault}(mid_1, fid_1, descr_1, ts_1) \wedge \text{Fault}(mid_2, fid_2, descr_2, ts_2) \\ & \wedge \text{Fix}(wid_3, fid_3, ts_3) && \text{if } fid_2 = fid_3 \ \&\& \ ts_2 > ts_1 + \text{TEN_MIN} \end{aligned}$$

where `TEN_MIN` is a Scala constant representing 10 minutes; for readability, similar constants are silently assumed throughout our examples. \square

3.2 Fair and Deterministic Matching Semantics for Join Patterns

We now define the notion of pattern matching for a join pattern $\Pi = \mu_1 \wedge \dots \wedge \mu_n \text{ if } \gamma$. If we have $n = 1$ and a single message m , we can apply standard definitions from functional programming languages [17] and say that Π matches m if there is a substitution σ such that (i) $\mu_1\sigma = m$, and (ii) $\gamma\sigma$ evaluates to `true`. (Clearly, such a match can only happen if σ substitutes *all* variables occurring in μ .) Instead, if $n > 1$ and we have multiple messages available in a mailbox M , things are more difficult: there may be multiple ways for the message pattern $\mu_1 \wedge \dots \wedge \mu_n$ to match different subsets of messages in M while satisfying the guard γ ; moreover, a join definition D might contain several join patterns that match (part of) the mailbox contents. Example 3.5 illustrates this.

► **Example 3.5** (Multiple options for join pattern matching). Let $D = \Pi_1 + \Pi_2$ where:

$$\begin{aligned}\Pi_1 &= \text{Fault}(id_1, _) \wedge \text{Fix}(id_2) \text{ if } id_1 = id_2 \\ \Pi_2 &= \text{Fault}(_, t_1) \wedge \text{Fault}(id_2, t_2) \wedge \text{Fix}(id_3) \text{ if } id_2 = id_3 \&& t_2 > t_1 + \text{TEN_MIN}\end{aligned}$$

(observe that D corresponds to the first two cases in Listing 1 modulo the omission of unused messages and variables). Suppose we have the following mailbox, where subscripts show the arrival order of messages (the lower the subscript, the older the message):

$$M = \text{Fault}_1(1, 10:35) \cdot \text{Fault}_2(2, 10:39) \cdot \text{Fault}_3(3, 10:56) \cdot \text{Fix}_4(3)$$

Before message Fix_4 lands in the mailbox, none of the join patterns matches any message combination in the mailbox. Instead, when Fix_4 arrives, we have the following options:

- the first join pattern matches the messages Fault_3 and Fix_4 , via the substitution $\{3, 3 / id_1, id_2\}$
- the second join pattern matches both:
 - messages Fault_1 , Fault_3 , and Fix_4 , via the substitution $\{3, 3, 10:35, 10:56 / id_1, id_3, t_1, t_2\}$;
 - messages Fault_2 , Fault_3 , and Fix_4 , via the substitution $\{3, 3, 10:39, 10:56 / id_1, id_3, t_1, t_2\}$. \sqcup

Existing implementations of join patterns leave the resolution of non-deterministic choices unspecified, or pick the first matching pattern as the “winner.” Our approach is different: we formalise a deterministic matching policy to give programmers control over the selection process. Consequently, we specify how to deterministically choose the messages matched by a join pattern in Example 3.5, as well as deterministically decide which join pattern “wins” when both match messages in the mailbox.

Def. 3.8 below formalises a *deterministic* and *fair* notion of join pattern matching: when a join pattern can match multiple combinations of messages in the mailbox, we prioritise the combination that consumes the oldest messages. To this end, we first introduce some notation in Def. 3.6 that we will use to reason about mailbox contents.

► **Definition 3.6** (Sequence length, indexing, slicing, and ordering). *Given a set S and a sequence $\mathcal{S} = s_1 \cdot \dots \cdot s_n$ containing elements of S , we write $|\mathcal{S}|$ for the length of \mathcal{S} , ϵ for the empty sequence, and $\mathcal{S}[i]$ with $i \in 1..|\mathcal{S}|$ for the element at the i^{th} position of \mathcal{S} . An indexing sequence, denoted by \mathcal{I} , is a non-empty sequence of pairwise-distinct natural numbers greater than 0. Given a sequence \mathcal{S} and an indexing sequence $\mathcal{I} = i_1 \cdot \dots \cdot i_n$ such that $i_h \in 1..|\mathcal{S}|$ for each $h \in 1..n$, we write $\mathcal{S}[\mathcal{I}]$ for the \mathcal{I} -slice of \mathcal{S} , which is the sequence $\mathcal{S}[i_1] \cdot \dots \cdot \mathcal{S}[i_n]$.*

Let S be a set with a total order \sqsubseteq . Then, the lexicographic order \leq_{lex} is the relation on sequences in S^ inductively defined as: (note that \leq_{lex} only relates sequences of equal length)*

$$\frac{}{\epsilon \leq_{\text{lex}} \epsilon} \quad \frac{s \sqsubseteq s' \quad s \neq s'}{s \cdot \mathcal{S} \leq_{\text{lex}} s' \cdot \mathcal{S}'} \quad \frac{\mathcal{S} \leq_{\text{lex}} \mathcal{S}'}{s \cdot \mathcal{S} \leq_{\text{lex}} s \cdot \mathcal{S}'}$$

Letting $\text{sort}(\mathcal{S})$ be the function returning the sorted sequence of elements of \mathcal{S} based on the total order \sqsubseteq , we also define the sorted length-biased lexicographic order \leq_{slex} as:

$$\frac{n = |\mathcal{S}'| \leq |\mathcal{S}| \quad \text{sort}(\mathcal{S})[1 \cdot \dots \cdot n] \leq_{\text{lex}} \text{sort}(\mathcal{S}')}{\mathcal{S} \leq_{\text{slex}} \mathcal{S}'}$$

We also define $=_{\text{slex}}$ as $\leq_{\text{slex}} \cap \leq_{\text{slex}}^{-1}$. Note that \leq_{slex} is a preorder.

► **Example 3.7.** Using relations \leq_{lex} and \leq_{slex} in Def. 3.6 on sequences of integers, we have:

$$\begin{aligned}1 \cdot 2 \cdot 3 &\leq_{\text{lex}} 1 \cdot 3 \cdot 2 \leq_{\text{lex}} 2 \cdot 1 \cdot 3 \leq_{\text{lex}} 2 \cdot 3 \cdot 1 \leq_{\text{lex}} 3 \cdot 1 \cdot 2 \leq_{\text{lex}} 3 \cdot 2 \cdot 1 \\ 1 \cdot 2 \cdot 3 \cdot 4 &=_{\text{slex}} 4 \cdot 3 \cdot 2 \cdot 1 <_{\text{slex}} 1 \cdot 2 \cdot 3\end{aligned}$$

► **Definition 3.8** (Fair join pattern matching). We define the following judgements

$$\begin{array}{ll} \mathcal{M} \models_{\sigma} \Pi & \text{the join pattern } \Pi \text{ exactly matches mailbox } \mathcal{M} \text{ via substitution } \sigma \\ \mathcal{M} \models_{\mathcal{I}} \Pi & \text{the join pattern } \Pi \text{ sparsely matches mailbox } \mathcal{M} \text{ via slice } \mathcal{I} \\ \mathcal{M} \models \Pi \rightsquigarrow \mathcal{I} & \text{the join pattern } \Pi \text{ fairly matches mailbox } \mathcal{M} \text{ via slice } \mathcal{I} \end{array}$$

by the following inference rules:

$$\frac{\forall i \in \{1, \dots, n\} : \mu_i \sigma = m_i \quad \gamma \sigma \quad \mathcal{M}[\mathcal{I}] \models_{\sigma} \Pi \quad \mathcal{M} \models_{\mathcal{I}} \Pi \quad \forall \mathcal{I}' : \mathcal{M} \models_{\mathcal{I}'} \Pi \implies \mathcal{I} \leq_{\text{lex}} \mathcal{I}'}{\mathcal{M} \models \Pi \rightsquigarrow \mathcal{I}}$$

In Def. 3.8, the judgement $\mathcal{M} \models_{\sigma} J$ if γ holds if J exactly matches all the messages in \mathcal{M} in the same order they occur therein, through a substitution σ such that the guard $\gamma \sigma$ holds.

This exact matching is used in the premise of judgement $\mathcal{M} \models_{\mathcal{I}} J$ if γ , which matches a slice \mathcal{I} of the mailbox \mathcal{M} : i.e., the message patterns in J may only match a (possibly reordered) subsequence $\mathcal{M}[\mathcal{I}]$ of the mailbox \mathcal{M} . Notice that the slice \mathcal{I} and the pattern J contain the same number of messages. Finally, the judgement $\mathcal{M} \models J$ if $\gamma \rightsquigarrow \mathcal{I}$ selects the smallest slice \mathcal{I} of \mathcal{M} w.r.t the order \leq_{lex} in Def. 3.6 such that $\mathcal{M} \models_{\mathcal{I}} J$ if γ holds. The selected slice \mathcal{I} represents the “fairest” possible match: \mathcal{I} indexes the oldest messages in \mathcal{M} that match J and satisfy the guard γ . This matching policy ensures that no message is left indefinitely in the mailbox if it can be used to match the join pattern. Note that, if two slices \mathcal{I} and \mathcal{I}' in the premise of the judgement contain the same indexes in different order (i.e., they may be deemed “equally fair”), the ordering \leq_{lex} deterministically selects the slice which minimises reordering between messages in \mathcal{M} and message patterns in J .

► **Example 3.9** (Fair join pattern matching). Let Π_1 , Π_2 , and \mathcal{M} be as in Example 3.5. By Def. 3.8 we have that:

- There is a single match for Π_1 : $\mathcal{M}[3 \cdot 4] \models_{\sigma} \Pi_1$ via $\sigma = \{3, 3 / id_1, id_2\}$. Hence, we also have $\mathcal{M} \models_{[3 \cdot 4]} \Pi_1$ and we also get $\mathcal{M} \models \Pi_1 \rightsquigarrow [3 \cdot 4]$ (i.e., the fairest way to match the join pattern Π_1 is to consume messages $Fault_3$ and Fix_4).
 - There are two matches for Π_2 :
 - $\mathcal{M}[1 \cdot 3 \cdot 4] \models_{\sigma} \Pi_2$ via $\sigma = \{3, 3, 10:35, 10:56 / id_1, id_3, t_1, t_2\}$. Therefore, $\mathcal{M} \models_{[1 \cdot 3 \cdot 4]} \Pi_1$;
 - $\mathcal{M}[2 \cdot 3 \cdot 4] \models_{\sigma} \Pi_2$ via $\sigma = \{3, 3, 10:39, 10:56 / id_1, id_3, t_1, t_2\}$. Therefore, $\mathcal{M} \models_{[2 \cdot 3 \cdot 4]} \Pi_2$.
- Hence, since $1 \cdot 3 \cdot 4 \leq_{\text{lex}} 2 \cdot 3 \cdot 4$, we conclude $\mathcal{M} \models \Pi_2 \rightsquigarrow [1 \cdot 3 \cdot 4]$ (i.e., the fairest way to match the join pattern Π_2 is to consume messages $Fault_1$, $Fault_3$, and Fix_4). \square

Def. 3.10 concludes this section by extending the notion of fair join pattern matching (Def. 3.8) to join definitions. The idea is that if a mailbox \mathcal{M} allows for multiple fair matches on different patterns in a join definition D , we pick the i^{th} join pattern in D that matches \mathcal{M} via the slice \mathcal{I} with the highest number of oldest messages w.r.t. the alternatives; and if two patterns in D yield equally fair matches, we pick the first in D . Since join patterns in D may match slices of different length, we rank the matches using \leq_{slex} (Def. 3.6). This approach makes the choice of patterns deterministic, while ensuring fairness.

► **Definition 3.10** (Matching of join definitions). The judgement $\mathcal{M} \models D \rightsquigarrow \mathcal{I}, i$ (read: “ D fairly matches mailbox \mathcal{M} via slice \mathcal{I} by its i^{th} join pattern”) is defined as:

$$\frac{\begin{array}{c} \text{Matches} = \{(\mathcal{I}, i) \mid i \in \{1, \dots, n\} \text{ and } \mathcal{M} \models \Pi_i \rightsquigarrow \mathcal{I}\} \\ (\mathcal{I}, i) \in \text{Matches} \quad \forall (\mathcal{I}', i') \in \text{Matches} : \mathcal{I} <_{\text{slex}} \mathcal{I}' \text{ or } (\mathcal{I} =_{\text{slex}} \mathcal{I}' \text{ and } i \leq i') \end{array}}{\mathcal{M} \models \Pi_1 + \Pi_2 + \dots + \Pi_n \rightsquigarrow \mathcal{I}, i}$$

► **Example 3.11** (Selecting the fairest match across join definitions (1)). Continuing Example 3.9, we can now apply Def. 3.10 to determine the fairest match for the join patterns sum $D = \Pi_1 + \Pi_2$. Since we have both:

$$\mathcal{M} \models \Pi_1 \rightsquigarrow [3 \cdot 4] \quad \text{and} \quad \mathcal{M} \models \Pi_2 \rightsquigarrow [1 \cdot 3 \cdot 4]$$

we rank the selected slices as $1 \cdot 3 \cdot 4 <_{\text{slex}} 3 \cdot 4$ (by Def. 3.6), i.e., the slice matched by Π_2 is fairer than the slice matched by Π_1 . Therefore, we conclude $\mathcal{M} \models D \rightsquigarrow (1 \cdot 3 \cdot 4), 2$. \square

► **Example 3.12** (Selecting the fairest match across join definitions (2)). To see why the relation \leq_{slex} (Def. 3.6) considers the lexicographical ordering of *sorted* sequences, consider this variation of Example 3.5:

$$\begin{aligned} \text{Fault}(id_1, _) \wedge \text{Fix}(id_2) && \text{if } id_1 = id_2 \\ + \quad \text{Fix}(id_3) \wedge \text{Fault}(_, t_1) \wedge \text{Fault}(id_2, t_2) && \text{if } id_2 = id_3 \ \& \& \ t_2 > t_1 + \text{TEN_MIN} \end{aligned}$$

Let Π_1 and Π_2 be the two join patterns above. Take the same mailbox \mathcal{M} used in Example 3.5. Observe that the message pattern constructors in Π_2 are reordered w.r.t. Example 3.5 – and therefore, we now have $\mathcal{M} \models \Pi_2 \rightsquigarrow [4 \cdot 1 \cdot 3]$ (i.e., the fairest match of Π_2 now consumes the slice $4 \cdot 1 \cdot 3$ of \mathcal{M}). Intuitively, this slice is lexicographically greater than the fairest slice $3 \cdot 4$ matched by Π_1 – but the slice $4 \cdot 1 \cdot 3$ consumes the older message at index 1. For this reason, by Def. 3.6 we have $4 \cdot 1 \cdot 3 =_{\text{slex}} 1 \cdot 3 \cdot 4 <_{\text{slex}} 3 \cdot 4$ – and consequently, the fairest match of Π_2 is ranked fairer than the fairest match of Π_1 . As a result, we obtain $\mathcal{M} \models D \rightsquigarrow (4 \cdot 1 \cdot 3), 2$ (by Def. 3.10) – i.e., despite the reordering of the message pattern constructors in Π_2 , we match the same messages of Example 3.11 (albeit with a differently-ordered slice). \square

3.3 Stateful, Tree-Based Matching Semantics for Join Patterns

Def. 3.8 offers a high-level specification for our notion of “fair matching” – but this definition does not automatically lead to an efficient implementation. To the contrary, the direct implementation of Def. 3.8 is a “stateless” brute-force algorithm that, whenever a new message reaches the mailbox: (i) enumerates all possible matches; (ii) lexicographically sorts the matches satisfying the guard γ , depending on the messages they use; and (iii) picks the match on the smallest mailbox slice (using the lexicographical ordering \leq_{lex} in Def. 3.6). This may require computing up to $n!$ message combinations for a mailbox of length n , every time a new message arrives. A similar brute-force approach is adopted in previous implementations of join patterns in literature [12]. A source of inefficiency is that many message combinations may be uselessly recomputed and retried when a new message arrives, even if such combinations have previously failed by falsifying the guard γ . Furthermore, when a new message yields two or more possible matches, finding the fairest one may lead to redundant computations. These issues are illustrated in Examples 3.13 and 3.14 below.

► **Example 3.13** (Redundant matching computations). Consider the join pattern Π_1 from Example 3.5 (recall that $\Pi_1 = \text{Fault}(id_1, _) \wedge \text{Fix}(id_2)$ if $id_1 = id_2$), and the following mailbox, where a message Fix_1 (emitted by a factory worker’s handheld device) is delivered to the monitor before the corresponding Fault_4 (emitted by a machine):

$$\mathcal{M} = \text{Fix}_1(3) \cdot \text{Fault}_2(1, 10:35) \cdot \text{Fault}_3(2, 10:36) \cdot \text{Fault}_4(3, 10:37)$$

We have to search for a match every time a new message lands in the mailbox:

- Π_1 cannot match message Fix_1 alone;
- when the message Fault_2 is delivered, the Π_1 matches $\text{Fix}_1 \cdot \text{Fault}_2$ – but the guard $id_1 = id_2$ is not satisfied;

17:12 Fair Join Pattern Matching for Actors

- when the message Fault_3 is delivered, the Π_1 can match $\text{Fix}_1 \cdot \text{Fault}_2 \cdot \text{Fault}_3$ in two possible ways using the combinations $(\text{Fix}_1, \text{Fault}_2)$ and $(\text{Fix}_1, \text{Fault}_3)$, neither of which satisfies the guard – note that $(\text{Fix}_1, \text{Fault}_2)$ has already been attempted;
- when the message Fault_4 is delivered, the Π_1 matches $\text{Fix}_1 \cdot \text{Fault}_2 \cdot \text{Fault}_3 \cdot \text{Fault}_4$ in a third possible way besides the previous two: in fact, the combination $(\text{Fix}_1, \text{Fault}_4)$ satisfies the guard – again note that the two failing combinations were already attempted. \square

► **Example 3.14** (Redundant fairness computations). Consider Π_2 and mailbox \mathcal{M} from Example 3.5:

$$\begin{aligned}\Pi_2 &= \text{Fault}(_, t_1) \wedge \text{Fault}(id_2, t_2) \wedge \text{Fix}(id_3) \text{ if } id_2 = id_3 \wedge t_2 > t_1 + \text{TEN_MIN} \\ \mathcal{M} &= \text{Fault}_1(1, 10:35) \cdot \text{Fault}_2(2, 10:39) \cdot \text{Fault}_3(3, 10:56) \cdot \text{Fix}_4(3)\end{aligned}$$

When the message Fix_4 lands in \mathcal{M} , the join pattern Π_2 matches two combinations of messages (as previously shown in Example 3.9), and they should be compared to determine the fairest. However, as soon as we determine that the combination $(\text{Fault}_1, \text{Fault}_3, \text{Fix}_4)$ satisfies the guard, it is redundant to try and compare other combinations – because none of them consumes the message Fault_1 , hence they cannot possibly be fairer, by Def. 3.8. \square

We present an algorithm to tackle these inefficiencies based on a *stateful* solution. Our algorithm keeps track of how the messages in a mailbox \mathcal{M} can *partially* match a join pattern Π , through a tree structure whose nodes contains sets of message indexes in \mathcal{M} , decorated with information on how such messages may complete Π . The way the tree is incrementally constructed allows us to (1) avoid recomputing previously-failed matches, and (2) immediately produce the fairest match (if it exists) via a depth-first traversal.

We use *mailbox trees* (*m-trees*) (Def. 3.15) to arrange integers (representing the indexes of the messages in a mailbox) into sets that form the nodes of a tree, so that, for each branch (X, Y) in the tree, the child node Y is a superset of its parent node X and $\max X < \max Y$.

► **Definition 3.15** (Mailbox trees). A mailbox tree on a finite set of natural number I (*m-tree* on I for short) is a tree $\mathcal{T} = (N, E)$ where:

- $N \subseteq 2^I$ is the set of nodes and $\emptyset \in N$ is the root the tree
- the cardinality of each node equals its level in \mathcal{T} and, for nodes X and Y at the same level, $X \cap Y = \emptyset$ and X precedes Y if $\max X \leq \max Y$
- for each edge $(X, Y) \in E$, $X \subset Y$ and $\max Y \notin X$.

We write $X \in \mathcal{T}$ if X is a node of \mathcal{T} and $i \in \mathcal{T}$ if there is $X \in \mathcal{T}$ such that $i \in X$. An *m-tree* \mathcal{T} on I is consistent when, for each level $h > 0$ of \mathcal{T} , $\bigcup\{X \in \mathcal{T} \mid X \text{ is at level } h\} = I$.

The “ramification” operation (Def. 3.16 below) is used to incrementally extend an m-tree by adding the index i of a new messages that has landed in a mailbox.

► **Definition 3.16** (Ramification). Given a tree $\mathcal{T} = (N, E)$ where the elements of N are subsets of numbers, and given a natural number $i \notin \mathcal{T}$, let:

$$N' = N \cup \{\bar{X} \cup \{i\} \mid \bar{X} \in N\} \quad \text{and} \quad E' = E \cup \{(Y \setminus \{\max Y\}, Y) \mid Y \in N'\}$$

Then, we call $r(\mathcal{T}, i) = (N', E')$ the ramification of \mathcal{T} with i .

Note that the ramification of a tree \mathcal{T} has twice as many nodes as \mathcal{T} . Also, the construction of m-trees does not depend on the order in which messages indexes are added, as shown in Proposition 3.17 below. This allows us to abbreviate $r(\dots r(\mathcal{T}, i_1), \dots, i_n)$ as $r(\mathcal{T}, \{i_1, \dots, i_n\})$.

► **Proposition 3.17.** Ramification is a commutative internal operation on m-trees.

Proof. That ramification is internal on m-trees follows by construction given how edges are extended in Def. 3.16. Commutativity follows by induction on the structure of $\textcolor{teal}{T}$. \blacktriangleleft

In Def. 3.20 below we decorate each node $\textcolor{brown}{X}$ in an m-tree with *assignments* that use the messages indexed by $\textcolor{brown}{X}$ to fill the variables in a join pattern. But first, we need some auxiliary constructions.

Given a mailbox \mathcal{M} and a join pattern $\textcolor{green}{J} \text{ if } \gamma$ with $\textcolor{green}{J} = \mu_1 \wedge \dots \wedge \mu_p$, we define the function $c : \{1, \dots, p\} \rightarrow 2^{\{1, \dots, |\mathcal{M}|\}}$ that maps each $i \in \{1, \dots, p\}$ to the set of indexes of messages in \mathcal{M} that match μ_i ; formally,

$$c(i) = \left\{ j \in 1..|\mathcal{M}| \mid \text{there is a substitution } \sigma \text{ such that } \mu_i \sigma = \mathcal{M}[j] \right\} \quad (1)$$

Also, an \mathcal{M} -assignment for $\textcolor{green}{J}$ is an injective map $a : \{1, \dots, p\} \rightarrow \{1, \dots, |\mathcal{M}|\}$ such that $i \in c(i)$ for all $1 \leq i \leq p$. Let $\text{asgn}(\mathcal{M}, \textcolor{green}{J})$ be the set of all \mathcal{M} -assignments for $\textcolor{green}{J}$. (Note that $\text{asgn}(\mathcal{M}, \textcolor{green}{J}) = \emptyset$ if $|\mathcal{M}| < p$.) The next Proposition 3.18 ensures that each assignment has a unique substitution induced by the matching; such a substitution can be effectively computed since the proof of Proposition 3.18 is constructive.

► **Proposition 3.18.** *For each $a \in \text{asgn}(\mathcal{M}, \mu_1 \wedge \dots \wedge \mu_p)$ there is a unique substitution σ_a such that $\mathcal{M}[a(i)] = \mu_i \sigma_a$, for all $i \in \{1, \dots, p\}$.*

Proof. Since each variable occurs at most once in $\mu_1 \wedge \dots \wedge \mu_p$ (by well-formedness, Def. 3.1), it suffices to take $\sigma_a = \bigcup_{i \in \{1, \dots, p\}} \sigma_i$ where $\mathcal{M}[a(i)] = \mu_i \sigma_i$ for all $i \in \{1, \dots, p\}$. \blacktriangleleft

An assignment $a \in \text{asgn}(\mathcal{M}, \textcolor{green}{J})$ is *valid for the guard* γ if $\gamma \sigma_a$ evaluates to *true* (with σ_a from Proposition 3.18).

► **Example 3.19** (Assignments). Let Π_1 and \mathcal{M} as in Example 3.13. We have $c(1) = \{2, 3, 4\}$ and $c(2) = \{1\}$, the assignment a such that $a(1) = 4$ and $a(2) = 1$ belongs to $\text{asgn}(\mathcal{M}, \text{Fault}(id_1, _) \wedge \text{Fix}(id_2))$ and it is valid for the guard $id_1 = id_2$ while for $a[3/1]$ (the update of a mapping 1 to 3) we have $a[1 \mapsto 3] \in \text{asgn}(\mathcal{M}, \text{Fault}(id_1, _) \wedge \text{Fix}(id_2))$ and $a[1 \mapsto 3]$ is not valid for $id_1 = id_2$. \dashv

We are now ready to introduce *assignment trees*.

► **Definition 3.20** (Assignment trees, pattern resolution). *The assignment tree of a join pattern $\textcolor{green}{J} \text{ if } \gamma$ w.r.t. mailbox \mathcal{M} is the pair $(\textcolor{teal}{T}, \textcolor{teal}{a})$ where, letting $I = c(\{1, \dots, p\})$ with c as in (1),*

- $\textcolor{teal}{T} = (N, E)$ *is the subtree up-to level p of $r((\emptyset, \emptyset), I)$, the m-tree on I and*
- *the map of candidate assignments $\textcolor{teal}{a} : N \rightarrow 2^{\text{asgn}(\mathcal{M}, \textcolor{green}{J})}$ is such that, for each node $\textcolor{brown}{X} \in N$,*

$$\textcolor{teal}{a}(\textcolor{brown}{X}) = \{a \in \text{asgn}(\mathcal{M}, \textcolor{green}{J}) \mid a \text{ is valid for } \gamma \text{ and } \text{cod } a = \textcolor{brown}{X}\}$$

Let $t(\mathcal{M}, \textcolor{green}{J} \text{ if } \gamma)$ denote the assignment tree of $\textcolor{green}{J} \text{ if } \gamma$ w.r.t \mathcal{M} . Pattern $\textcolor{green}{J}$ is resolved in \mathcal{M} if there is a leaf $\textcolor{brown}{X}$ in $t(\mathcal{M}, \textcolor{green}{J} \text{ if } \gamma)$ such that $\textcolor{teal}{a}(\textcolor{brown}{X}) \neq \emptyset$.

Proposition 3.21 below is a soundness result: the assignment tree for mailbox \mathcal{M} and join pattern Π only contains combinations of message indexes from \mathcal{M} that can contribute to matching Π . Instead, Theorem 3.22 is a completeness result: it says that if an assignment matches a join pattern Π for mailbox \mathcal{M} , then the m-tree of \mathcal{M} contains a node made of exactly the messages used by that assignment. Taken together, these two results guarantee that, if we inspect assignment trees to find possible matches for Π in mailbox \mathcal{M} , we can only find possible matches (soundness), and we will not miss any possible match (completeness).

► **Proposition 3.21.** *The assignment tree $t(\mathcal{M}, \Pi)$ is consistent.*

Proof. The union of nodes of the same level but 0 yields $c(\{1, \dots, p\})$. ◀

► **Theorem 3.22.** *For all $a \in \text{asgn}(\mathcal{M}, J)$, $\text{cod } a \in t(\mathcal{M}, J \text{ if } \gamma)$.*

Proof. Assume $J = \mu_1 \wedge \dots \wedge \mu_p$ and proceed by induction on p using Def. 3.16. ◀

We now need to rank the assignments in an m-tree to align with our “fair matching” policy (Def. 3.8). To this end, we define the total order \preceq on $\text{asgn}(\mathcal{M}, J)$ as follows:

$$a \preceq a' \quad \text{if} \quad \langle a(1) \cdot \dots \cdot a(p) \rangle \leq_{\text{lex}} \langle a'(1) \cdot \dots \cdot a'(p) \rangle \quad \text{where} \quad J = \mu_1 \wedge \dots \wedge \mu_p \quad (2)$$

Now, Def. 3.23 below formalises how a join pattern is “fairly” resolved in an assignment tree. Observe that, crucially, Def. 3.23 only considers the first node in a depth-first traversal of the assignment tree that yields some candidate assignments. This allows our algorithm to find the fairest matches first, and avoid unnecessary traversals.

► **Definition 3.23** (Fair resolution). *Let X be the first node in a depth-first visit of the assignment tree $t(\mathcal{M}, J \text{ if } \gamma)$ at level p whose candidate assignment map a is non-empty. The fair resolution of $t(\mathcal{M}, J \text{ if } \gamma)$ is the minimal assignment in $a(X)$ w.r.t pre-order \preceq in (2).*

Note that Def. 3.23 univocally identifies a fair resolution when a join pattern matches multiple slices, as shown in Example 3.24 below.

► **Example 3.24.** Let Π_1 be as in Example 3.5 and consider the mailbox:

$$\text{Fault}_1(3, 10:35) \cdot \text{Fault}_2(2, 10:39) \cdot \text{Fault}_3(3, 10:56) \cdot \text{Fix}_4(3)$$

The assignments $a = \begin{cases} 1 \mapsto 1 \\ 2 \mapsto 4 \end{cases}$ and $a' = \begin{cases} 1 \mapsto 3 \\ 2 \mapsto 4 \end{cases}$ are valid (observe that $\sigma_a \neq \sigma_{a'}$), and their fair resolution is a , since $a \preceq a'$. □

Finally, Theorem 3.25 below shows that the tree-based algorithm correctly computes the fair join pattern matching according to Def. 3.8.

► **Theorem 3.25.** *Let $\Pi = J \text{ if } \gamma$ with $J = \mu_1 \wedge \dots \wedge \mu_p$, then $\mathcal{M} \models \Pi \rightsquigarrow \mathcal{I}$ if and only if the fair resolution a of $t(\mathcal{M}, \Pi)$ is such that $\mathcal{I} = a(1) \cdot \dots \cdot a(p)$.*

Proof. (\implies) Let $\mathcal{M}[\mathcal{I}] = m_1 \cdot \dots \cdot m_p$. By Def. 3.8, there is substitution σ such that $\gamma\sigma$ holds and $\mu_i\sigma = m_i$ for all $i \in \{1, \dots, p\}$ and any other slice with this property is greater than \mathcal{I} . Let a be the assignment such that $a(i) = \mathcal{I}[i]$ for all $i \in \{1, \dots, p\}$. By construction, $a \in \text{asgn}(\mathcal{M}, J)$, hence $\text{cod } a \in t(\mathcal{M}, J \text{ if } \gamma)$ by Theorem 3.22. Let a' be the resolution of $t(\mathcal{M}, J \text{ if } \gamma)$. By definition $a' \preceq a$ which, by (2), is equivalent to say that $\langle a'(1) \cdot \dots \cdot a'(p) \rangle \leq_{\text{lex}} \langle a(1) \cdot \dots \cdot a(p) \rangle$. We then have the thesis since $a = a'$ because we also have $\langle a(1) \cdot \dots \cdot a(p) \rangle \leq_{\text{lex}} \langle a'(1) \cdot \dots \cdot a'(p) \rangle$ by hypothesis.

(\impliedby) Let \mathcal{I}' be such that $\mathcal{M} \models \Pi \rightsquigarrow \mathcal{I}'$, map a be the resolution of $t(\mathcal{M}, \Pi)$, and $\mathcal{I}' = [a(1) \cdot \dots \cdot a(p)]$. We have $\mathcal{M} \models_{\mathcal{I}'} \Pi$ since $\mathcal{M}[\mathcal{I}] \models_{\sigma_a} \Pi$ by construction. Therefore $\mathcal{I}' \leq_{\text{lex}} \mathcal{I}$ by Def. 3.8 and the codomain of the assignment a' such that $a'(i) = \mathcal{I}'[i]$ for all $i \in \{1, \dots, p\}$ belongs to $t(\mathcal{M}, \Pi)$ by Theorem 3.22. Let $\mathcal{Y} \in t(\mathcal{M}, \Pi)$ containing this codomain and $X \in t(\mathcal{M}, \Pi)$ be such that $\text{cod } a \in X$. We have the following cases: either X precedes \mathcal{Y} , or \mathcal{Y} precedes X , or else $X = \mathcal{Y}$. In the first case, if $\max X < \max \mathcal{Y}$ then $\mathcal{I}' \neq \mathcal{I}$ and $\mathcal{I}' \leq_{\text{lex}} \mathcal{I}$, which contradicts the hypothesis $\mathcal{M} \models \Pi \rightsquigarrow \mathcal{I}$. In the second case, if $\max \mathcal{Y} < \max X$ then $\mathcal{I}' \neq \mathcal{I}$ and $\mathcal{I} \leq_{\text{lex}} \mathcal{I}'$, which contradicts the fact that a' is the resolution of $t(\mathcal{M}, \Pi)$. It must therefore be $X = \mathcal{Y}$ which implies $a' = a$. □

4 Implementation: the JoinActors Library

We now present `JoinActors`, our actor-based implementation of join patterns and fair matching algorithms in the Scala 3 programming language. `JoinActors` is the companion artifact of this paper, and its latest version is available at:

<https://github.com/a-y-man/join-actors>

In Section 4.1 we provide an overview of the join patterns API, and the main components of the library and motivation behind the choice of Scala 3. In Section 4.2 we present the implementation of the stateful tree-based matching, and in Section 4.3 we describe our prototype actor framework with fair join pattern matching.

4.1 Overview

The API of `JoinActors` allows programmers to write join patterns using (almost-)standard Scala pattern-matching syntax (as shown in the code snippet in Listing 1); at compile-time, the pattern matching code is transformed (using metaprogramming) into an internal data structure that is used by the matching algorithms to perform the join pattern matching. To use the library, the programmer calls the `receive` function (which is actually a macro) passing join patterns as regular Scala 3 pattern-matching expressions. This function also take as a parameter the type of matching algorithm to use. The syntax for join definitions is:

```
receive { (self: ActorRef[...]) =>
  case J1 if γ1 => RHS1
  case J2 if γ2 => RHS2 ... }
```

We selected Scala 3 for our join pattern library because its macros allow us to implement a straightforward API for join patterns, without necessitating specialized syntax or compiler extensions. This way, programmers can write join patterns using familiar language constructs, eliminating the need to learn a new language or syntax.

Our library uses a `Matcher` trait as a common interface to two matching algorithms:

- `BruteForceMatcher`: the brute-force matching algorithm; this is a translation of the declarative semantics with no state management, described in Section 3.2;
- `StatefulTreeMatcher`: the stateful tree-based matching algorithm described in Section 3.3.

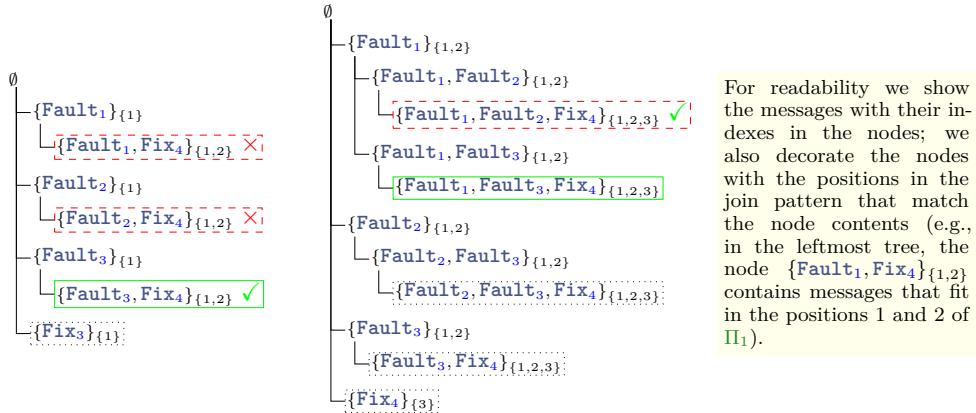
4.2 Implementing Stateful Tree-based Matching

M-trees (Def. 3.15) are the basic data structure of the our algorithm which are the cornerstone to build assignment trees (Def. 3.20). Given a join pattern $\Pi = J \text{ if } \gamma$ with $J = \mu_1 \wedge \dots \wedge \mu_p$ and a mailbox \mathcal{M} , our implementation *lazily* builds $t(\mathcal{M}, \Pi)$ using the ramification operation (Def. 3.16) starting from the tree (\emptyset, \emptyset) and incrementally extending it in depth-first order. When the messages of \mathcal{M} indexed by a leaf X at level p complete Π , we check the guard γ :

- if γ is satisfied by an assignment induced by X , we report the match, and remove the matched messages from \mathcal{M} and from the assignment tree (stopping its ramification);
- otherwise, we optimise the tree by pruning the leaf X , and continue its ramification.

If no match is found, we wait for a new message to land in the mailbox, and repeat.

► **Example 4.1** (Assignment tree construction). The assignment trees for Π_1 , Π_2 , and \mathcal{M} in Example 3.5 are pictorially shown below:



Leaves yielding a successful match (i.e., a combination of messages that complete the message pattern and satisfy its guard) are marked with green solid box and symbol \checkmark . Leaves where messages completes the join pattern without satisfying its guard are marked with red dashed boxes and symbol \times ; such leaves are immediately pruned from the tree once computed. Leaves in dotted boxes are other *potential* message matches – which are *not* actually computed, because an earlier successful match is found while lazily ramifying the tree (and the earlier match is fairer than the later potential match). \square

In the implementation, the M-trees are represented using the `TreeMap`⁴ data structure: it is a sorted map that takes an ordering on the keys, and for the ordering we use Def. 3.6. We use the following data structure types:

- `type PatternBins = TreeMap[PatternIdxs, MessageIdxs]` is a map from the positions of the patterns to the indices of the messages that match the pattern. These are the subscripts of the nodes in the trees of Example 4.1, where we associate the index of a message to the index of the pattern matching it. If a join pattern contains several messages with the same constructor (such as `Fault` in Π_2 in Example 4.1) then these messages will be grouped in the same bin where the key is the sequence of indices of the join pattern and the value will be the sequence of indices of the matched messages from the mailbox. For instance, the pattern bin of the leaf node with green solid box $\{Fault_1, Fault_2, Fix_4\}_{\{1,2,3\}}$ in the rightmost tree in Example 4.1 would be represented as $[1, 2] \mapsto [1, 2], [3] \mapsto [4]$.
- `type MatchingTree = TreeMap[MessageIdxs, PatternBins]` is a map from the indices of the messages that have been matched so far to the `PatternBins`. Thus, the leaf node with green solid box in the rightmost tree in Example 4.1 would have the matching tree $[1, 2, 4] \mapsto [[1, 2] \mapsto [1, 2], [3] \mapsto [4]]$.

Note that these data structures contain only the indices of the partially-matched messages and patterns. The guard is checked only when a leaf node is completed (as described above).

4.3 Prototype Actor Framework

To showcase our join patterns implementation in the actor concurrency model, `JoinActors` offers a prototype actor framework in Scala. Notably, our implementation requires Java 21 or later to run due to the use of *virtual threads*. We use `LinkedTransferQueues` as the mailbox implementation, and `ActorRef` objects for sending messages into a mailbox.

⁴ <https://scala-lang.org/api/3.x/scala/collection/immutable/TreeMap.html>

```

1 class Actor[M, T](private val matcher: Matcher[M, Result[T]]):
2   private val mailbox = LinkedTransferQueue[M]
3   val self           = ActorRef(mailbox)
4
5   def start(): (Future[T], ActorRef[M]) =
6     val promise = Promise[T]
7     ec.execute(() => run(promise))
8     (promise.future, self)
9
10  @tailrec
11  private def run(promise: Promise[T]): Unit =
12    matcher(mailbox)(self) match
13      case Continue    => run(promise)
14      case Stop(value) => promise.success(value)

```

■ **Listing 2** The `Actor` class implementation in the `JoinActors` library.

In Listing 2 (line 1) the actor’s constructor takes a `Matcher` instance as a parameter. The `run` method processes the messages in the mailbox using the `matcher` instance built by the `receive` macro, which may be either a `BruteForceMatcher` or a `StatefulTreeMatcher` instance. Depending on the result of the right-hand side of the join pattern, the actor either continues processing messages or stops and returns a result.

5 Evaluation

In this section we present the evaluation of our implementation of join patterns and the matching algorithms. In Section 5.1 we describe the methodology used to evaluate the performance of the algorithms. In Section 5.2 we present the results of the custom synthetic benchmarks. In Section 5.3 we present the results of a smart house benchmark. In Section 5.4 we present the results of a bounded buffer benchmark. In Section 5.5 we analyse the correlation between the size of the actor mailbox and the size of the m-trees. In Section 5.6 we compare our implementation of the tree-based fair matching algorithm with an alternative implementation based on the RETE algorithm. Overall, our experiments show that:

- Our stateful tree-based algorithm outperforms the brute force strategy in “noisy” workloads, where messages forming the fairest match are interspersed with random and non-matching messages. On the contrary, when the messages for the fairest match arrive consecutively, the brute force strategy outperforms the stateful tree-based one, since the latter incurs the overhead of building the matching tree. It is worth noticing that non-noisy workloads are unlikely in distributed or asynchronous scenarios.
- Compared to a RETE-based implementation of fair matching, our tree-based algorithm avoids unnecessary production of matches (as it directly picks the “fairest match”) and scales better when guards are computationally heavy. However, our tree-based algorithm implementation is an unoptimised proof-of-concept, and for simple guards may perform worse than an optimised RETE implementation (as the one we used).
- The number of matches per second measured in our experiments also shows that our implementation of the stateful tree-based matching is suitable for both our floor shop opening example and the smart house scenario – where the expected input traffic is in the order of tens of messages per second, with moderate “noise.”

5.1 Methodology

The lack of a standard benchmark suite for join patterns makes the performance evaluation non-trivial, as the performance of matching algorithms is highly sensitive to the inputs (i.e. amount and order of incoming messages), the size of the message patterns, and the complexity of the guards. Similar sensitivity has been documented e.g. in [15], which compares the matching algorithms RETE [6] and TREAT [18] in the realm of expert and rule-based multi-agent systems. Given that our stateful tree-based algorithm is influenced by RETE and TREAT, our evaluation methodology is inspired by the assessment conducted in [15].

We have devised a set of benchmarks to draw insights into the performance comparison between our stateful tree-based algorithm and the naïve brute-force algorithm. We also identify the trade-offs between the two algorithms in different scenarios, i.e., the overhead of maintaining state versus the overhead of reprocessing messages. To this end, we have created custom synthetic benchmarks and adapted some benchmarks from the literature, such as a producers-consumers bounded buffer from the Savina benchmark suite [13] and a smart house benchmark adapted from [22]. We ran the experiments on a computer with dual Xeon E5-2687W (8-core, 3.10GHz) and 128GB of memory, with 64-bit GNU/Linux 5.10.27. We used OpenJDK 21 with default settings, maximum heap size set to 16 GB, and Scala 3.3.3.

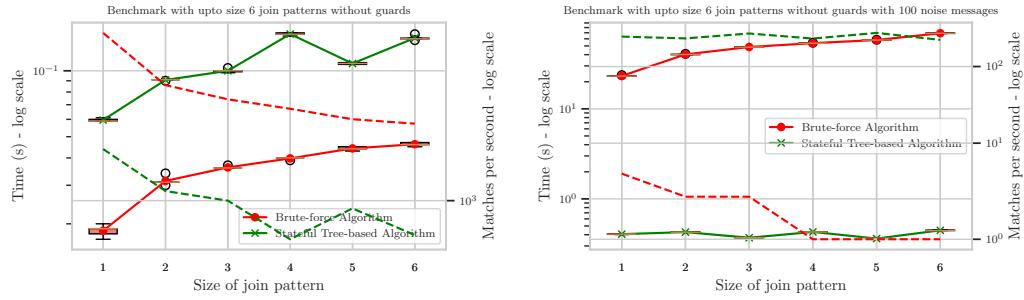
5.2 Synthetic Benchmarks

The general setup of each benchmark involves a program with a single actor, which consists of precisely one join definition, which receives message sent incrementally without delays. A benchmark execution finishes once the actor has processed all matches for the messages contained in its mailbox. We start measuring time just before the first message is sent and stop when the actor halts, so to disregard the time for setting up the actor. The benchmarks are implemented in Scala 3, and are included in the companion artifact of this paper.

Each experiment is repeated 5 times. The plots in Figs. 1 and 2 are to be read as follows: the x -axis represents the join pattern size (i.e., the number of messages in the pattern), and the error bars show the standard deviation of the measurements; the solid lines (measured on the left y -axis, log scale) show the benchmark completion time; the dashed lines (measured on the right y -axis, log scale) show the number of matches per second.

The benchmark suite has been crafted to encompass the following three aspects.

1. **Pattern size.** We have considered actors with pattern sizes ranging from 1 to 6: this mirrors scenarios found in the literature, where join patterns are usually not very long.
2. **Message workload profile.** We have benchmarked two kinds of input traffic workload: (1) a “clean” workload where the messages delivered to the mailbox precisely match the join pattern; (2) a spectrum of “noisy” workloads, where varying amounts of messages delivered to the mailbox will not match any pattern. The noise is uniformly interspersed with matchable messages. The rationale behind this choice is that the first scenario should favour the brute-force algorithm, as it may minimize the advantages of maintaining state, allowing us to measure the overhead of state maintenance. Conversely, the second scenario will require the brute-force algorithm to analyse unusable combinations of messages, thereby enabling us to measure the benefits of maintaining state.
3. **Guard effect.** We evaluated actors with join patterns, with and without guards. The inclusion of patterns with guards is particularly tailored to the main goal of this paper, which is to assess the benefits of state-based algorithms in the presence of guards.



■ **Figure 1** Pattern size without guards benchmark: without noise (left) and with noise (right).

5.2.1 Pattern Size and Workload without Guards

The first group of benchmarks compares the performance of brute-force and tree-based algorithms in cases where actors do not use guards. We consider actors with the following shape, for size 5 (i.e., a unique join pattern matching case for 5 messages). Note that messages have no payload, and the only rule has no guard.

```

1 Actor[SizeMsg, ...] {
2   receive { (_: ActorRef[SizeMsg]) =>
3     { case (A(), B(), C(), D(), E()) => ... }
4   } }
```

The corresponding benchmark evaluates the performance of such actor when fed with a stream of messages consisting on 100 repetitions of the sequence `A()`, `B()`, `C()`, `D()`, `E()`. Note that the mailbox is fed with messages in the exact order required for the match.

The results of the benchmark, considering actors of size 1 to 6, for both algorithms are shown in Fig. 1. The plot on the left of Fig. 1 shows that the brute-force approach significantly outperforms the stateful tree-based approach because the latter has the inherent overhead to build and to update trees – whereas the brute-force algorithm defers processing until a sufficient number of messages are received. Due to the nature of the traffic sent to the actor, the brute-force algorithm immediately finds a match every n messages, where n is the size of the pattern (e.g., 5 in the code snippet above). Instead, the stateful tree-based algorithm has to update its tree for each message, and only after n messages will it find a match and then prune the tree: hence, the retained state is only marginally utilised. However, as shown in the right plot of Fig. 1, if we change the shape of the messages sent to the actor, by augmenting the sequence of messages with noise (i.e. messages that do not match the pattern), the stateful tree-based algorithm outperforms the brute-force algorithm.

5.2.2 Pattern Size and Workload with Guards

The next benchmark addresses the effect of guards. Actors resemble the ones in Section 5.2.1, but now messages have payload and join patterns are augmented with guards, as follows:

```

1 Actor[SizeMsg, ...] {
2   receive { (_: ActorRef[SizeMsg]) => {
3     case (A(x), B(y), C(z), D(w), E(a))
4       if x == y && y == z && z == w && w == a => ... }
5   } }
```

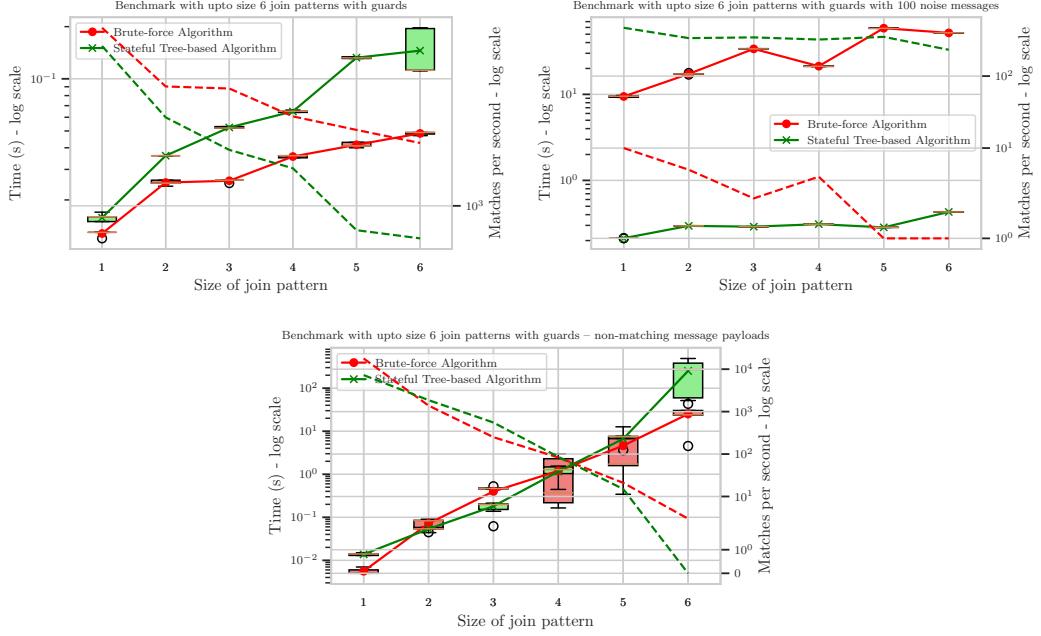


Figure 2 Benchmark results for varying pattern sizes with guards: without noise (top left), and with additional noise messages (top right) that cannot be matched by any join pattern. The bottom plot is a variation where the noise consists of messages that might be potentially matched by the join patterns, but whose payloads do not satisfy their guard.

As before, we first address the case of “perfect” workload, by feeding the actor with sequences of messages as `A(1)`, `B(1)`, `C(1)`, `D(1)`, `E(1)`, `A(2)`, `B(2)`, `C(2)`, `D(2)`, `E(2)`,

The results of the benchmark are shown in Fig. 2 (top-left). Similarly to the benchmark without guards, the brute-force algorithm outperforms the stateful tree-based algorithm on well-behaved input traffic. However, when we augment the sequence of “noise” messages, the results are similar to the benchmark without guards. Namely, the stateful tree-based algorithm outperforms the brute-force algorithm. This can be seen in Fig. 2 (top-right).

Moreover, we have performed a variation of this benchmark with a different type of noise: this time, the sequence of messages sent to the actor is augmented with payloads that do *not* satisfy the guard. An example of such a sequence of messages is:

`A(1)`, `A(-3)`, `B(1)`, `B(-4)`, `C(1)`, `C(-5)`, `D(1)`, `D(-6)`, `E(1)`, `E(-7)`, `A(2)`, ...

where only the messages highlighted in green match the pattern.

The benchmark results are shown in Fig. 2 (bottom). The performance of both algorithms is similar, which aligns with our expectations. Noise messages will always be reprocessed by both algorithms, as they persist as partial matches in the m-tree and as unprocessed messages in the brute-force algorithm. These noise messages can only be discarded if they satisfy the guard condition, which is not the case in this benchmark.

5.3 Smart House Benchmark

This benchmark is a real-world scenario adapted from [22]. In this setup, a single actor represents the smart house monitor and controller, tasked with managing various smart devices within a household. Specifically, the actor (1) activates the lights when someone enters and the ambient light is below 40 lux, and (2) detects arrivals or departures based on specific message sequences and reacts accordingly. The actor is shown in Listing 3.

```

1 Actor[Action, ...] {
2   receive { (selfRef: ActorRef[Action]) =>
3     case (
4       Motion(_: Int, mStatus: Boolean, mRoom: String, t0: Date),
5       AmbientLight(_: Int, value: Int, alRoom: String, t1: Date),
6       Light(_: Int, lStatus: Boolean, lRoom: String, t2: Date)
7     ) if bathroomOccupied(...) => ...
8     case (
9       Motion(_: Int, mStatus0: Boolean, mRoom0: String, t0: Date),
10      Contact(_: Int, cStatus: Boolean, cRoom: String, t1: Date),
11      Motion(_: Int, mStatus1: Boolean, mRoom1: String, t2: Date)
12    ) if occupiedHome(...) => ...
13     case (
14       Motion(_: Int, mStatus0: Boolean, mRoom0: String, t0: Date),
15       Contact(_: Int, cStatus: Boolean, cRoom: String, t1: Date),
16       Motion(_: Int, mStatus1: Boolean, mRoom1: String, t2: Date)
17     ) if emptyHome(...) => ...
18   ...
19 }
```

Listing 3 Smart house actor (arguments of predicates in the guards are omitted for readability). The type annotations in the join patterns are not necessary, but they are included for clarity.

Each of the mentioned scenarios is implemented as a separate join pattern with guards to ensure the correct behaviour is triggered. The performance evaluation of the implementation is conducted by measuring the time taken to process a number of messages that activate the patterns 1000 times, interspersed with a number of random messages intended to mimic various real-world workloads. When the number of random messages is 0, the smart house actor receives only exact matches: thus, for a join pattern size of 3, the actor will process 3000 such messages to get 1000 matches. Instead, in the case of 16 random messages, the actor receives 1000 sequences of messages, each one consisting of 16 random messages plus 3 matching messages distributed throughout the sequence. The benchmark concludes once the smart house actor has performed the expected 1000 matches.

Fig. 3 shows the results of our experiments: our implementation of the stateful tree-based algorithm quickly outperforms the naïve brute-force one, as it can quickly reuse partial matches and discard failed matches – whereas the brute-force algorithm has to compute all possible matches for each new incoming message. The plot also shows that the tree-based matching algorithm performs $\sim 10^5$ matches/sec. on non-noisy traffic (i.e., every input message is used in a join pattern match), and degrades to $\sim 10^2$ as more noise is injected in the input traffic, until $\sim 90\%$ of the messages are noise. This suggests that the implementation is suitable for a real-world application where a smart house controller may expect tens of messages per second with some amount of noise.

5.4 Producers-Consumers Bounded Buffer

This benchmark is an example of a multi-process synchronization problem. The benchmark involves producers and consumers represented by actors and a buffer actor. The buffer actor acts as a manager: (1) it monitors whether the data buffer is full or empty; (2) when consumers request work from an empty buffer, they are placed in a queue until work becomes

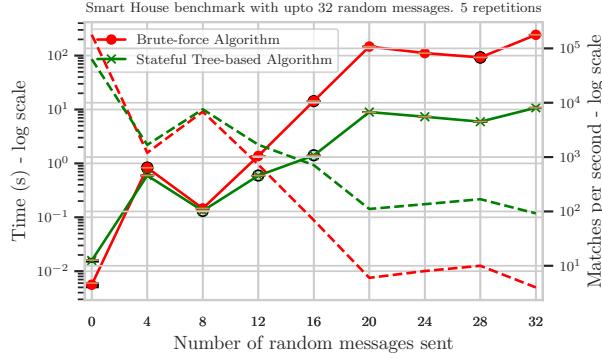


Figure 3 Smart House Benchmark. The x -axis represents the number of random “noise” messages sent with each group of 3 matchable messages; the solid lines (measured on the left y -axis, log scale) show the completion time; the dashed lines (measured on the right y -axis, log scale) show the number of matches per second. For each data point, the smart house actor performs 1000 matches, and the benchmark is repeated 5 times. The error bars show the standard deviation of the measurements.

available; (3) when producers are prepared to produce data but the buffer is full, they are queued until space opens up in the buffer; (4) it alerts producers to generate more work when space becomes available in the data buffer.

Fig. 4 shows the results of the benchmark for a buffer size of 1000. Since the join patterns of the bounded buffer are simple and without any guards and the messages are well-behaved, the brute-force algorithm outperforms the stateful tree-based algorithm. The overhead of maintaining state in the stateful tree-based algorithm is not justified in this case.

5.5 Analysis of Mailbox Size vs. Match Tree Size

We now focus on the relationship between the size of mailboxes and the size of the matching trees maintained in-memory by our stateful matching algorithm. Our analysis uses the smart house benchmark according to three different workload scenarios:

- **No noise:** each batch of 3 incoming messages triggers a match, emptying the mailbox after each match.
- **50% noise:** each batch consists of 3 messages suitable for matching and 3 “noise” messages that cannot be used in the matching, and thus, just accumulate in the mailbox.
- **66% noise:** each batch consists of 3 messages suitable for matching and 6 “noise” ones. For each scenario, we send 10 batches of messages, triggering 10 join pattern matches. The results are collected in Fig. 5, where the plots in the each row correspond to one of our scenarios. Despite the potential for m-trees to grow exponentially with mailbox size, the plots show a mostly flat and almost-linear correlation between mailbox size and m-tree size. For instance, the left plot in the first row of Fig. 5 illustrates the relationship between the number of messages processed by the actor and both the mailbox and m-tree sizes. Spikes indicate a join pattern match, leading to message removal and m-tree pruning. The right plot in the first row further explores the correlation between mailbox and m-tree sizes, demonstrating, for example, that a mailbox with two messages correlates to an m-tree size of 5-7 nodes, depending on the messages and applicable join patterns.

The results for the noise scenarios are respectively in the second and third row of Fig. 5: they show that mailbox and m-tree sizes decrease after each match similarly to the no-noise scenario. However, “noise” messages accumulate generating partial matches in the m-tree. Flat lines in the m-tree size plot signify unmatchable messages, leaving m-trees unchanged.

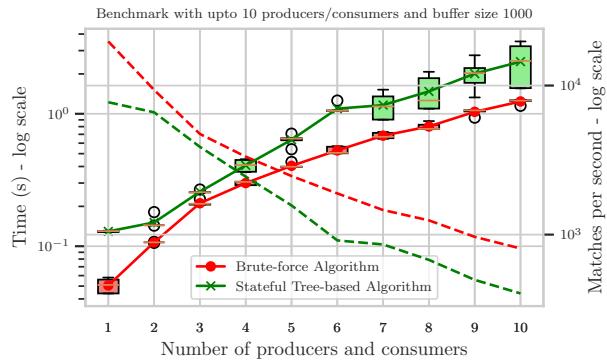


Figure 4 Producers-consumers bounded buffer benchmark: time to produce and consume data in a buffer of size 1000 against the number producers and consumers. The solid lines (measured on the left y -axis, log scale) show the benchmark completion time; the dashed lines (measured on the right y -axis, log scale) show the number of matches per second. The benchmark is repeated 5 times. The error bars show the standard deviation of the measurements.

5.6 Comparison with a RETE-based Fair Matching Implementation

In this section we compare our implementation of our stateful tree-based fair join pattern matching algorithm against an implementation based on the RETE algorithm. The use of RETE takes inspiration from [28], but here we use the Evrete library for Java.⁵ To implement an Evrete-based actor that realises our fair matching policy, we proceed as follows.

1. We set up an Evrete *session* where:
 - each incoming message is modelled as a *fact* with a unique id (denoting the order of arrival), and
 - each pattern matching case is encoded as a *rule* that Evrete will check against all combinations of “message facts” in the session. If a combination of “message facts” satisfies a rule, their message ids are stored in a collection of matches for that rule.
2. We implement an actor (as a JVM virtual thread) that, when a new message arrives:
 - a. stores the message as a “message fact” in the aforementioned Evrete session,
 - b. fires the session rules, and
 - c. if one or more successful matches are produced by any of the rules, then:
 - i. sorts the collections of successful matches (if any) by fairness (using Def. 3.6 on the “message fact” ids);
 - ii. finds the fairest match;
 - iii. removes from the session all the “message facts” used by such a fairest match; and
 - iv. clears the collections of successful matches.

A key difference between RETE and our stateful tree-based fair matching algorithm is that RETE exhaustively computes *all* possible matches when the rules are fired, and such matches must be sorted to find the fairest (see item 2(c)i above). Instead, our algorithm only computes matches “on demand” by finding the fairest first, through a lazy depth-first traversal of the match tree. This suggests that our tree-based fair matching algorithm is computationally less expensive than the RETE-based implementation outlined above. However, the comparative benchmarks are also influenced by multiple implementation differences:

⁵ <https://www.evrete.org>

17:24 Fair Join Pattern Matching for Actors

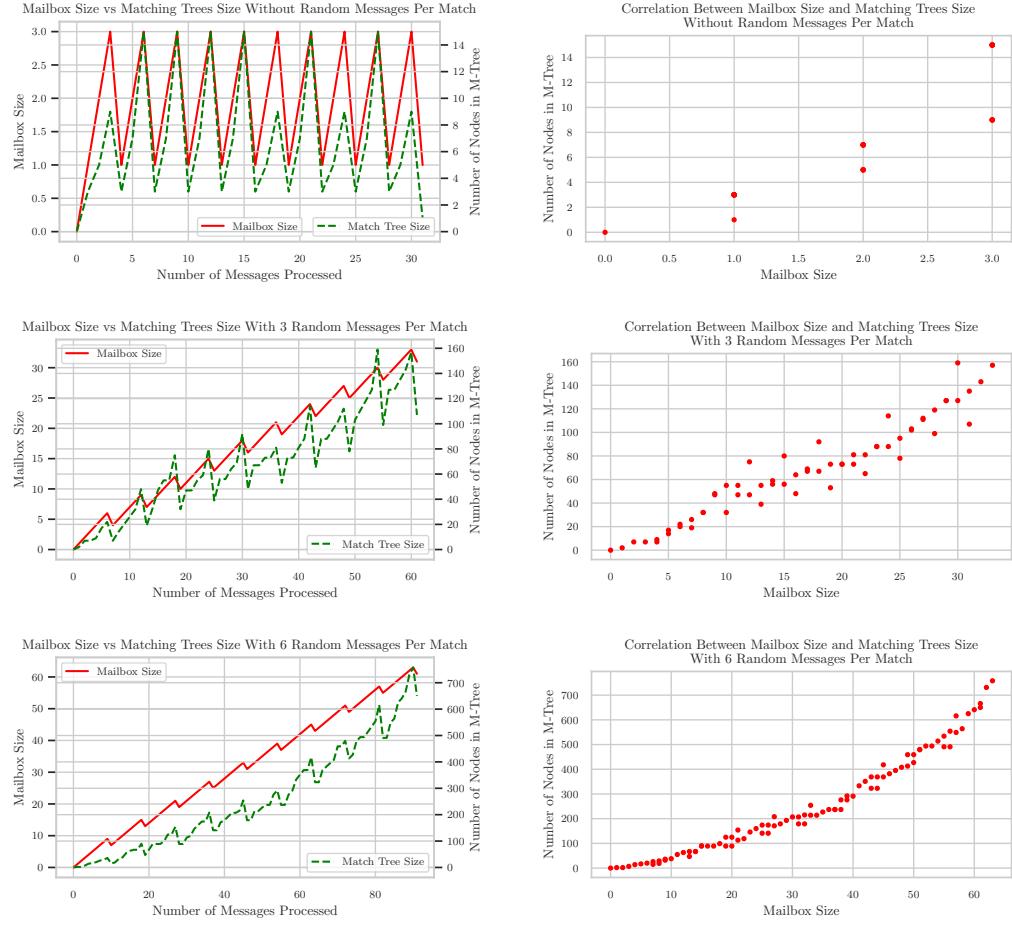


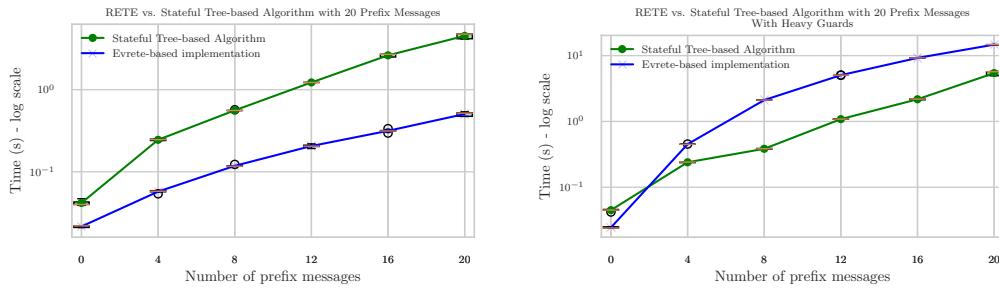
Figure 5 Mailbox size against the size of the matching tree: sizes based on the number of processed messages (left column) and mailbox/tree size correlation (right column).

- our implementation of tree-based fair matching is a proof-of-concept, is not optimised, is single-threaded, and is almost completely written in functional Scala. In particular, adding partial matches to the m-tree is a rather expensive operation, because the m-tree is currently implemented as an immutable data structure;
- instead, Evrete is being developed since 2020 and is significantly optimised, multi-threaded, and written in imperative Java using high-performance mutable data structures.

Consequently, Evrete can produce a significantly higher amount of matches-per-second w.r.t. our implementation – and thus, its exhaustive production of matches can be often faster than our “on-demand, fairest-first” production.

For these reasons, we have designed a benchmark (based on the “smart house” in Section 5.3) that discriminates the computational characteristics of our tree-based fair matching algorithm and the Evrete-based implementation, despite implementation differences:

1. we send n groups of “prefix” messages that create a partial match for a join pattern;
2. then, we send one message that can complete the join pattern with any of the previous “prefix” messages.



■ **Figure 6** Comparison of tree-based vs. Evrete-based implementation of fair matching. The x -axis shows the number of groups of “prefix” messages sent before a completing message. The y -axis shows the time taken to perform 10 matches.

E.g., for the pattern `Motion(...)` \wedge `AmbientLight(...)` \wedge `Light(...)`, we send n times the “prefix” messages `Motion(...)`, `AmbientLight(...)` (which partially match the pattern), and finally we send a message `Light(...)`, which can be combined with any previous “prefix” message to complete the pattern. In this situation, the Evrete-based implementation will compute all possible matches and then find the fairest – while our stateful tree-based fair matching algorithm will immediately produce the fairest match between `Light(...)` and the oldest `Motion(...)`, `AmbientLight(...)` messages. The benchmark measures the time taken to process up to n groups of messages followed by a completing message. In total, we send $(2n+1) \times 10$ messages, thus triggering 10 matches. We perform two variants of this benchmark:

- one with simple guards (the ones used in Section 5.3), and
- one where we artificially slow down the time necessary to evaluate the guards, simulating computationally-intensive “heavy guards” that take ~ 0.1 milliseconds to be computed.

The results are shown in Fig. 6. The plot on the left (with “simple” guards) shows that the Evrete-based implementation of fair matching outperforms our stateful tree-based implementation. The plot on the right shows that, with “heavy guards”, our stateful tree-based implementation outperforms the Evrete-based one: this is because our tree-based algorithm evaluates the “heavy guards” only once (after finding the fairest match), whereas Evrete computes the “heavy guards” repeatedly, for each possible match contained in the actor mailbox. Observe that the overall execution time of our algorithm in the plots of Fig. 6 does not change significantly. This suggest that our algorithm is less sensitive to “heavy guards” than the Evrete-based implementation.

6 Conclusion

We have addressed the problem of formalising and implementing join pattern matching for actor-based concurrent and distributed systems. We have contributed a novel specification of *fair and deterministic join pattern matching* guaranteeing that the oldest messages in an actor’s mailbox are eventually consumed, if allowed by the patterns. We have presented a novel *stateful tree-based join pattern matching algorithm* that avoids wasteful recomputations across matches, and we have proved that our algorithm correctly implements the fair matching specification. We have presented a novel actor library for Scala 3 that implements fair join pattern matching, with both stateless and stateful algorithms. We have presented a systematic benchmark suite for join-pattern-based systems, and we have applied it to evaluate our implementation. We have assessed the performance of our stateful tree-based algorithm in

comparison to the brute-force algorithm and a RETE-based implementation, under various conditions. The findings reveal a performance trade-off: the brute-force algorithm excels when dealing with well-behaved workloads, where maintaining state is an unnecessary overhead, whereas the stateful tree-based algorithm outperforms in scenarios with relative noise in the input messages (which is to be expected in many real-world applications) and complex guards, as evidenced in the smart house benchmark. The synthetic benchmarks in Sections 5.2.1 and 5.2.2 underscore the high sensitivity of the matching algorithms to their workload and guard complexity. These insights should be taken into account when choosing the matching algorithm, depending on the nature of the application and anticipated workload.

Future work. Our specification of fair join pattern matching includes several design decisions that may be fine-tuned depending on the application context. E.g., in some settings it may be better to select the pattern with the longest match, and in some settings it may be useful to match the *newest* messages in a mailbox (e.g., if the input traffic volume is too high for processing every message in real-time). We can specify and implement these alternative policies with minimal adjustments to our definitions, results, and library implementation: we plan to study them, and explore other possibilities. Also, ensuring “fair choice” when multiple join patterns are enabled is another intriguing notion of fairness that would require a significantly different formalisation of matching semantics; we leave this as future work.

We plan to study the problem of *join pattern unreachability*, i.e., whether a pattern will always be preempted by its alternatives. E.g., if a join definition contains the two join patterns $A(x) + A(x) \wedge B(y)$, the former may be always preferred to the latter, or not, depending on the nuances of the matching policy across patterns (e.g., first-match vs. longest-match). We plan to study how to statically verify whether a join pattern is unreachable, and extend our Scala 3 implementation to issue a compile-time warning when this occurs.

Our evaluation shows that selecting the best-performing strategy for join pattern matching is not trivial: depending on the expected input traffic and the complexity of the patterns and guards, stateful matching may be faster than stateless matching, or *vice versa*. Our Scala 3 library `JoinActors` can be easily tweaked to allow programmers choose a specific matching strategy *per pattern*; it could also be extended to switch matching strategy “on the fly” (i.e., between matches), and it could be interesting to study how to automatically switch strategy depending on input traffic observations. We plan to adapt `JoinActors` to let programmers select a suitable matching strategy based on a custom heuristic.

`JoinActors` is a proof-of-concept prototype, and we plan to heavily optimise it – in particular, by using a more efficient mutable data structure to represent m-trees, allowing for faster updates when new messages arrive, and faster traversals for finding the fairest match. The need for such optimisations is highlighted by the results shown in Section 5.6, and we plan to study the internals of the Evrete library to inspire future improvements.

We see our evaluation in Section 5 as a first necessary step towards establishing a standard, comprehensive benchmark suite for existing and future join pattern implementations, in the spirit of Savina [13] for actor implementations. We will study how to further improve our benchmark suite, and we welcome feedback and suggestions from the community.

References

- 1 Gul A. Agha. ACTORS - a model of concurrent computation in distributed systems. In *MIT Press series in artificial intelligence*, 1986.
- 2 Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004. doi:10.1145/1018203.1018205.

- 3 Oliver Braćevac, Nada Amin, Guido Salvaneschi, Sebastian Erdweg, Patrick Eugster, and Mira Mezini. Versatile event correlation with algebraic effects. *Proc. ACM Program. Lang.*, 2(ICFP), July 2018. doi:10.1145/3236762.
- 4 Mariangiola Dezani, Roland Kuhn, Sam Lindley, and Alceste Scalas. Behavioural Types: Bridging Theory and Practice (Dagstuhl Seminar 21372). *Dagstuhl Reports*, 11(8):52–75, 2022. doi:10.4230/DagRep.11.8.52.
- 5 Fabrice Le Fessant and Luc Maranget. Compiling join-patterns. *Electronic Notes in Theoretical Computer Science*, 16(3):205–224, 1998. doi:10.1016/S1571-0661(04)00143-4.
- 6 Charles Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. *Expert Systems*, pages 324–341, 1991.
- 7 Cédric Fournet, Fabrice Le Fessant, Luc Maranget, and Alan Schmitt. Jocaml: A language for concurrent distributed and mobile programming. In Johan Jeuring and Simon L. Peyton Jones, editors, *Advanced Functional Programming, 4th International School, AFP 2002, Oxford, UK, August 19-24, 2002, Revised Lectures*, volume 2638 of *Lecture Notes in Computer Science*, pages 129–158. Springer, 2002. doi:10.1007/978-3-540-44833-4_5.
- 8 Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In Hans-Jürgen Boehm and Guy L. Steele Jr., editors, *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 372–385. ACM Press, 1996. doi:10.1145/237721.237805.
- 9 Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *CONCUR'96: Concurrency Theory: 7th International Conference Pisa, Italy, August 26–29, 1996 Proceedings* 7, pages 406–421. Springer, 1996.
- 10 Rosita Gerbo and Luca Padovani. Concurrent Typestate-Oriented Programming in Java. In Francisco Martins and Dominic Orchard, editors, *Proceedings Programming Language Approaches to Concurrency- and Communication-cEtric Software, PLACES@ETAPS 2019, Prague, Czech Republic, 7th April 2019*, volume 291 of *EPTCS*, pages 24–34, 2019. doi:10.4204/EPTCS.291.3.
- 11 Rob Van Glabbeek and Peter Höfner. Progress, justness, and fairness. *ACM Comput. Surv.*, 52(4), August 2019. doi:10.1145/3329125.
- 12 Philipp Haller and Tom Van Cutsem. Implementing joins using extensible pattern matching. In Doug Lea and Gianluigi Zavattaro, editors, *Coordination Models and Languages, 10th International Conference, COORDINATION 2008, Oslo, Norway, June 4-6, 2008. Proceedings*, volume 5052 of *Lecture Notes in Computer Science*, pages 135–152. Springer, 2008. doi:10.1007/978-3-540-68265-3_9.
- 13 Shams M. Imam and Vivek Sarkar. Savina - an actor benchmark suite: Enabling empirical evaluation of actor libraries. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control, AGERE! '14*, pages 67–80, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2687357.2687368.
- 14 G. Stewart Von Itzstein and Mark Jasiunas. On implementing high level concurrency in Java. In Amos Omondi and Stanislav Sedukhin, editors, *Advances in Computer Systems Architecture, 8th Asia-Pacific Conference, ACSAC 2003, Aizu-Wakamatsu, Japan, September 23-26, 2003, Proceedings*, volume 2823 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2003. doi:10.1007/978-3-540-39864-6_13.
- 15 Eryk Lagun. Evaluation and implementation of match algorithms for rule-based multi-agent systems using the example of jadex. *MSc Thesis, University of Hamburg*, 2009. URL: https://swa.informatik.uni-hamburg.de/files/abschlussarbeiten/Diplomarbeit_Eryk_Lagun.pdf.
- 16 Qin Ma and Luc Maranget. Compiling pattern matching in join-patterns. In *International Conference on Concurrency Theory*, pages 417–431. Springer, 2004. doi:10.1007/978-3-540-28644-8_27.

- 17 Luc Maranget. Compiling pattern matching to good decision trees. In Eijiro Sumii, editor, *Proceedings of the ACM Workshop on ML, 2008, Victoria, BC, Canada, September 21, 2008*, pages 35–46. ACM, 2008. doi:10.1145/1411304.1411311.
- 18 Daniel P. Miranker. *TREAT: a new and efficient match algorithm for AI production systems*. PhD thesis, Columbia University, USA, 1987. UMI Order No. GAX87-10209.
- 19 Martin Odersky. Functional nets. In *European Symposium on Programming*, pages 1–25. Springer, 2000. doi:10.1007/3-540-46425-5_1.
- 20 Hubert Plociniczak and Susan Eisenbach. Erlang with joins. In *Coordination Models and Languages: 12th International Conference, COORDINATION 2010, Amsterdam, The Netherlands, June 7-9, 2010. Proceedings 12*, pages 61–75. Springer, 2010.
- 21 Humberto Rodriguez Avila. *Orchestration of Actor-Based Languages for Cyber-Physical Systems*. PhD thesis, Vrije Universiteit Brussel, 2021.
- 22 Humberto Rodríguez-Avila, Joeri De Koster, and Wolfgang De Meuter. Advanced join patterns for the actor model based on CEP techniques. *Art Sci. Eng. Program.*, 5(2):10, 2021. doi:10.22152/programming-journal.org/2021/5/10.
- 23 Claudio V. Russo. The Joins concurrency library. In Michael Hanus, editor, *Practical Aspects of Declarative Languages, 9th International Symposium, PADL 2007, Nice, France, January 14-15, 2007*, volume 4354 of *Lecture Notes in Computer Science*, pages 260–274. Springer, 2007. doi:10.1007/978-3-540-69611-7_17.
- 24 Claudio V. Russo. Join patterns for visual basic. 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 53–72. ACM, 2008. doi:10.1145/1449764.1449770.
- 25 Antoine Louis Thibaut Sébert. Join-patterns for the actor model in Scala 3 using macros. Master’s thesis, DTU Department of Applied Mathematics and Computer Science, 2022. Available at <https://findit.dtu.dk/en/catalog/62f83d3680aa6403a4ccc0ab>.
- 26 Martin Sulzmann, Edmund S. L. Lam, and Peter Van Weert. Actors with multi-headed message receive patterns. In Doug Lea and Gianluigi Zavattaro, editors, *Coordination Models and Languages, 10th International Conference, COORDINATION 2008, Oslo, Norway, June 4-6, 2008. Proceedings*, volume 5052 of *Lecture Notes in Computer Science*, pages 315–330. Springer, 2008. doi:10.1007/978-3-540-68265-3_20.
- 27 Aaron Joseph Turon and Claudio V. Russo. Scalable join patterns. 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 575–594. ACM, 2011. doi:10.1145/2048066.2048111.
- 28 Louise Van Verre, Humberto Rodríguez-Avila, Jens Nicolay, and Wolfgang De Meuter. Florence: A hybrid logic-functional reactive programming language. *Proceedings of the 9th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, 2022.

A CFL-Reachability Formulation of Callsite-Sensitive Pointer Analysis with Built-In On-The-Fly Call Graph Construction

Dongjie He¹  

University of New South Wales, Sydney, Australia
Chongqing University, China

Jingbo Lu¹  

University of New South Wales, Sydney, Australia
Shanghai Sectrend Information Technology Co., Ltd, China

Jingling Xue  

University of New South Wales, Sydney, Australia

Abstract

In object-oriented languages, the traditional CFL-reachability formulation for k -callsite-sensitive pointer analysis (k CFA) focuses on modeling field accesses and calling contexts, but it relies on a separate algorithm for call graph construction. This division can result in a loss of precision in k CFA, a problem that persists even when using the most precise call graphs, whether pre-constructed or generated on the fly. Moreover, pre-analyses based on this framework aiming to improve the efficiency of k CFA may inadvertently reduce its precision, due to the framework's lack of native call graph construction, essential for precise analysis.

Addressing this gap, this paper introduces a novel CFL-reachability formulation of k CFA for Java, uniquely integrating on-the-fly call graph construction. This advancement not only addresses the precision loss inherent in the traditional CFL-reachability-based approach but also enhances its overall applicability. In a significant secondary contribution, we present the first precision-preserving pre-analysis to accelerate k CFA. This pre-analysis leverages selective context sensitivity to improve the efficiency of k CFA without sacrificing its precision. Collectively, these contributions represent a substantial step forward in pointer analysis, offering both theoretical and practical advancements that could benefit future developments in the field.

2012 ACM Subject Classification Theory of computation → Program analysis

Keywords and phrases Pointer Analysis, CFL Reachability, Call Graph Construction

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.18

Supplementary Material Software (Artifact): <https://doi.org/10.5281/zenodo.11061892> [15]

Funding ARC Grants DP230102871, DP240103194, and the Fundamental Research Funds for the Central Universities of Ministry of Education of China (No. 2024CDJXY015).

Acknowledgements We thank the anonymous reviewers for their constructive comments.

1 Introduction

Pointer analysis is fundamental to numerous static analyses, including program understanding, program verification, security analysis, compiler optimization, and symbolic execution. Over the past two decades, k -callsite-sensitivity [49], which distinguishes method contexts on their k -most-recent callsites, has emerged as a prevalent context abstraction in both whole-program [5, 60, 40] and demand-driven [53, 48, 62] pointer analyses for Java programs.

¹ The first two authors contributed equally to this work.

Traditionally, k -callsite-sensitive pointer analysis, abbreviated to k CFA (Control-Flow Analysis) [49], is either inclusion-based [1] or founded on context-free language (CFL) reachability [44]. The inclusion-based formulation for k CFA [22, 57] has been incorporated into several pointer analysis frameworks for Java [40, 59, 60, 5, 17]. In this approach, a program’s statements are represented as points-to set constraints. The methods’ calling contexts are delineated by parameterizing these constraints with context abstractions. Often, the call graph for the program is constructed dynamically, i.e., on the fly to maximize precision and efficiency [11, 47, 26, 27, 50]. Conversely, the CFL-reachability formulation for k CFA [53] plays a pivotal role in the development of a diverse array of pointer analysis algorithms. These include demand-driven pointer/alias analysis [53, 64, 62, 48], context transformations [57], library-code summarization [48], and selective context-sensitivity [33]. In this approach, a program’s points-to information is determined by resolving a graph reachability problem within a specifically constructed *pointer assignment graph* (PAG) [26]. This CFL-reachability formulation involves analyzing the intersection of two context-free languages (CFLs), denoted as $L_{FC} = L_F \cap L_C$, where L_F describes field accesses as balanced parentheses and L_C enforces callsite-sensitivity by matching method calls and returns, also represented through balanced parentheses [53]. However, this formulation employs a distinct, external algorithm for call graph construction, further elaborated in Section 2.

In comparison to the inclusion-based approach, the L_{FC} -based CFL-reachability formulation for k CFA suffers from two major limitations, primarily due to its reliance on a separate algorithm for call graph construction. Firstly, this segregation can lead to a decrease in precision within k CFA, a problem that persists regardless of whether the call graphs are pre-constructed or generated on the fly. Secondly, certain pre-analyses, such as SELECTX [33], aim to enhance k CFA’s efficiency through the L_{FC} -based CFL-reachability formulation. However, these pre-analyses might unintentionally compromise its precision, undermining the overall effectiveness of the pointer analysis.

The primary contribution of this research lies in addressing the aforementioned limitations by introducing a new CFL-reachability formulation of k CFA. This novel formulation, for the first time, demonstrates the feasibility of specifying k CFA entirely through CFL-reachability, eliminating the need for a separate call graph algorithm. Our approach utilizes three CFLs, $L_{DCR} = L_D \cap L_C \cap L_R$, within a new PAG framework. Here, L_D extends beyond field accesses (as in L_F) to include dynamic dispatch, L_C maintains callsite-sensitivity as per previous formulation [53], and L_R introduces support for parameter passing required by its built-in on-the-fly call graph construction. Theoretically, we demonstrate for the first time that k CFA can be characterized as a specific type of context-sensitive language – the intersection of multiple CFLs. This is a notable distinction, as not all context-sensitive languages can be expressed in this manner [31, 25], underscoring the uniqueness of our approach. The subsequent sections will delve into the challenges of designing L_{DCR} and provide insights into our formulation’s underpinnings.

As a secondary contribution of this research, we demonstrate the practical utility of L_{DCR} by introducing P3CTX, the first precision-preserving pre-analysis designed to accelerate k CFA in Java programs. Given the critical importance of precision in tasks such as software security analysis, our approach distinguishes itself as the preferable option. It provides a speed advantage without sacrificing precision. P3CTX employs an L_{DCR} -enabled selective context-sensitivity technique, further substantiating the correctness of L_{DCR} . In contrast, SELECTX [33], developed based on L_{FC} [53], invariably encounters precision loss, thus underscoring the superiority of our approach.

$$\begin{array}{c}
 \frac{x = \mathbf{new} T // O \quad ctx \in \text{MethodCtx}(M)}{\langle O, [ctx]_h \rangle \in \text{PTS}(x, ctx)} \text{ [I-NEW]} \quad \frac{x = y \quad ctx \in \text{MethodCtx}(M)}{\text{PTS}(y, ctx) \subseteq \text{PTS}(x, ctx)} \text{ [I-ASSIGN]} \\
 \\
 \frac{x = y.f \quad ctx \in \text{MethodCtx}(M)}{\langle O, htx \rangle \in \text{PTS}(y, ctx)} \text{ [I-LOAD]} \quad \frac{x.f = y \quad ctx \in \text{MethodCtx}(M)}{\langle O, htx \rangle \in \text{PTS}(x, ctx)} \text{ [I-STORE]} \\
 \\
 \frac{x = m(a_1, \dots, a_n) // c \quad ctx \in \text{MethodCtx}(M) \quad ctx' = [c :: ctx]_k}{\begin{aligned} &ctx' \in \text{MethodCtx}(m) \quad \text{PTS}(\text{ret}^m, ctx') \subseteq \text{PTS}(x, ctx) \\ &\forall i \in [1, n] : \text{PTS}(a_i, ctx) \subseteq \text{PTS}(p_i^m, ctx') \end{aligned}} \text{ [I-SCALL]} \\
 \\
 \frac{x = r.m(a_1, \dots, a_n) // c \quad ctx \in \text{MethodCtx}(M) \quad \langle O, htx \rangle \in \text{PTS}(r, ctx) \quad t = \text{DynTypeOf}(O) \quad m' = \text{dispatch}(c, t) \quad ctx' = [c :: ctx]_k}{\begin{aligned} &ctx' \in \text{MethodCtx}(m') \quad \text{PTS}(\text{ret}^{m'}, ctx') \subseteq \text{PTS}(x, ctx) \\ &\langle O, htx \rangle \in \text{PTS}(\text{this}^{m'}, ctx') \quad \forall i \in [1, n] : \text{PTS}(a_i, ctx) \subseteq \text{PTS}(p_i^{m'}, ctx') \end{aligned}} \text{ [I-VCALL]}
 \end{array}$$

Figure 1 Inclusion-based formulation (M is the containing method of the statement being analyzed).

In summary, this paper makes the following two major contributions:

- A new CFL-reachability formulation of *kCFA* with built-in call graph construction.
- An *L_{DCR}*-enabled precision-preserving pre-analysis for accelerating *kCFA* with selective context-sensitivity. Compared with two state-of-the-art pre-analyses [33, 29], our pre-analysis enables better efficiency-precision trade-offs in several application scenarios.

The rest of this paper is organized as follows. Section 2 provides background knowledge and motivates the development of *L_{DCR}* by highlighting several design challenges. Section 3 introduces *L_{DCR}*, explaining how these challenges are addressed and offering insights into its design. Section 4 presents and evaluates, P3CTX, our *L_{DCR}*-enabled pre-analysis for accelerating *kCFA*. Section 5 discusses related work and Section 6 concludes the paper.

2 Background and Motivation

We start by reviewing the inclusion-based and traditional CFL-reachability *L_{FC}* formulations of *kCFA* (Section 2.1). Next, we use an example to illustrate their approaches to call graph construction, discuss *L_{FC}*'s limitations, and highlight the necessity of and challenges faced in designing *L_{DCR}*, a new CFL-reachability formulation with an integrated on-the-fly call graph construction (Section 2.2).

In our formalization, we consider a simplified Java language with six types of statements: New for object creation (“`x = new T // O`”); Assign for variable assignments (“`x = y`”); Load for retrieving field values (“`x = y.f`”); Store for assigning values to fields (“`x.f = y`”); Virtual Calls for instance method calls (“`x = r.m(a1, ..., an) // c`”); and Static Calls for static method calls (“`x = m(a1, ..., an) // c`”). Here, O identifies the unique abstract object created by a particular New statement, x and y are local variables, and c identifies a callsite. For a virtual call `r.m(a1, ..., an)`, we write `thism'`, `pim'` and `retm'` as its “this” variable, *i*-th parameter and return variable for a virtual method *m'* invoked at this callsite, respectively. For a static call `m(a1, ..., an)`, only `pim` and `retm` are relevant. In scenarios where method calls do not return a value, the flow from `retm` to x is disregarded.

$$\begin{array}{c}
\frac{x = \mathbf{new} \ T \ // \ O}{O \xrightarrow{\text{new}} x} [\mathbf{P-NEW}] \quad \frac{x = y}{y \xrightarrow{\text{assign}} x} [\mathbf{P-ASSIGN}] \quad \frac{x = y.f}{y \xrightarrow{\text{load}[f]} x} [\mathbf{P-LOAD}] \\
\\
\frac{x.f = y}{y \xrightarrow{\text{store}[f]} x} [\mathbf{P-STORE}] \quad \frac{x = m(a_1, \dots, a_n) \ // \ c}{\forall i \in [1, n] : a_i \xrightarrow[\hat{c}]{\text{assign}} p_i^m \quad \text{ret}^m \xrightarrow[\check{c}]{\text{assign}} x} [\mathbf{P-SCALL}] \\
\\
\frac{x = r.m(a_1, \dots, a_n) \ // \ c \quad m' \text{ is a target of this callsite}}{r \xrightarrow[\hat{c}]{\text{assign}} \text{this}^{m'} \quad \text{ret}^{m'} \xrightarrow[\check{c}]{\text{assign}} x \quad \forall i \in [1, n] : a_i \xrightarrow[\hat{c}]{\text{assign}} p_i^{m'}} [\mathbf{P-VCALL}]
\end{array}$$

Figure 2 Rules for building the PAG required by L_{FC} .

2.1 Background

2.1.1 Inclusion-based Formulation

Figure 1 gives the rules for such a formulation [22, 51, 57], where several auxiliary functions are used: (1) `MethodCtx` maintains the contexts used for analyzing a method, (2) `dispatch` resolves a virtual call to a target method, and (3) `PTS` records the points-to information found context-sensitively for a variable or an object’s field. In k CFA, context sensitivity is achieved by parameterizing variables and objects with contexts as modifiers. A calling context of a method is abstracted by its last k callsites. Given a context $ctx = [c_1, \dots, c_n]$ and a context element c , $c :: ctx$ stands for $[c, c_1, \dots, c_n]$ and $[ctx]_k$ stands for $[c_1, \dots, c_k]$.

Let us examine the six rules in Figure 1. In **[I-NEW]**, hk represents the (heap) context length for a heap object, typically set as $hk = k - 1$ [51, 58, 20, 30]. **[I-ASSIGN]**, **[I-LOAD]**, and **[I-STORE]** address standard assignments and field accesses. **[I-SCALL]** and **[I-VCALL]** handle static and virtual calls, respectively. Let us explain **[I-VCALL]** only. In this rule, m' is a target method dynamically resolved for a receiver object O (based on its dynamic type $t = \text{DynTypeOf}(O)$) at callsite c . Thus, this rule is also responsible for performing on-the-fly call graph construction during the pointer analysis. In its conclusion, $ctx' \in \text{MethodCtx}(m')$ reveals how the contexts of a method are introduced. Initially, for the program being analyzed, its entry methods have only the empty context, e.g., $\text{MethodCtx}(\text{"main"}) = \{[]\}$. Importantly, the receiver variable r and the other arguments a_1, \dots, a_n are handled differently: a receiver object flows only to the method it dispatches, while the objects pointed to by $a_i (i \in [1, n])$ flow to all methods dispatched at this callsite.

2.1.2 L_{FC} -based CFL-Reachability Formulation

In L_{FC} [53], k CFA is solved by reasoning about CFL-reachability on a PAG representation [26]. Figure 2 gives six rules for building the PAG. For a PAG edge, its label above indicates whether it is an assignment or field access. There are two types of `assign` edges: *intra-procedural* (for modeling regular assignments without a below-edge label) and *inter-procedural* (for modeling parameter passing with a below-edge label representing a callsite).

In L_{FC} , passing arguments to parameters at both static and virtual callsites is modeled identically by using inter-procedural `assign` edges (**[P-SCALL]** and **[P-VCALL]**). For example, in **[P-VCALL]**, \hat{c} (\check{c}) signifies an inter-procedural value-flow entering into (exiting from) m' at callsite c , where m' represents a virtual method discovered by a separate call graph construction algorithm (either in advance [9, 2, 55] or on the fly [54, 53]). Therefore, \hat{c} (\check{c}) is also known as an *entry* (*exit*) *context*.

```

1 class A {
2   void foo(D p) {
3     Object v = p.f;
4   }
5 }
6 class B extends A {
7   void foo(D q) { }
8 }
9 class C extends A {
10  void foo(D r) {}
11 }
12 class D { Object f; }
13 class O { }

14 static void bar(A x, O o) {
15   D d = new D(); // D1
16   d.f = o;
17   x.foo(d); // c3
18 }
19 static void main() {
20   O o1 = new O(); // O1
21   O o2 = new O(); // O2
22   A a = new A(); // A1
23   A b = new B(); // B1
24   bar(a, o1); // c1
25   bar(b, o2); // c2
26 }

```

Figure 3 A motivating example.

For a PAG edge $x \xrightarrow[c]{\ell} y$, its *inverse edge*, which is omitted in Figure 2 but required by L_{FC} , is defined as $y \xrightarrow[\bar{c}]{\bar{\ell}} x$. For a below-edge label \hat{c} or \check{c} , $\bar{\hat{c}} = \check{c}$ and $\bar{\check{c}} = \hat{c}$, implying that the concepts of entry and exit contexts for inter-procedural assign edges are swapped if they are traversed inversely.

L_{FC} is defined as the intersection of two distinct CFLs, $L_{FC} = L_F \cap L_C$, with L_F pertaining to the PAG's above-edge labels and L_C to its below-edge labels. L_F , a CFL over Σ_{L_F} , is created from above-edge labels. For each path p in the PAG, $L_F(p)$ is a string in $\Sigma_{L_F}^*$, made by sequentially concatenating p 's above-edge labels. A node v is L_F -reachable from node u if a path p , termed L_F -path, exists from u to v such that $L_F(p) \in L_F$. L_C follows a similar definition, but with Σ_{L_C} comprising below-edge labels.

We give L_F and L_C below and illustrate both with an example in Section 2.2. L_F enforces field-sensitivity for field accesses by matching stores and loads as balanced parentheses:

$$\begin{array}{lcl}
\text{flowsto} & \longrightarrow & \text{new flows}^* \\
\text{flows} & \longrightarrow & \underline{\text{assign}} \mid \text{store}[f] \text{ alias load}[f] \\
\text{alias} & \longrightarrow & \text{flowsto flowsto} \\
\underline{\text{flowsto}} & \longrightarrow & \overline{\text{flows}}^* \text{ new} \\
\text{flows} & \longrightarrow & \underline{\text{assign}} \mid \text{load}[f] \text{ alias store}[f]
\end{array} \tag{1}$$

Note that u alias v iff $u \xrightarrow{\text{flowsto}} O \xrightarrow{\text{flowsto}} v$ for some object O . In addition, $O \xrightarrow{\text{flowsto}} v$ iff $v \xrightarrow{\text{flowsto}} O$, meaning that flowsto actually represents the standard points-to relation.

L_C enforces callsite-sensitivity by matching “calls” and “returns” as balanced parentheses:

$$\begin{array}{lcl}
\text{realizable} & \longrightarrow & \text{exit entry} \\
\text{exit} & \longrightarrow & \text{exit balanced} \mid \text{exit } \check{c} \mid \epsilon \\
\text{entry} & \longrightarrow & \text{entry balanced} \mid \text{entry } \hat{c} \mid \epsilon \\
\text{balanced} & \longrightarrow & \text{balanced balanced} \mid \hat{c} \text{ balanced } \check{c} \mid \epsilon
\end{array} \tag{2}$$

A path p in the PAG of the program is *realizable* iff p is an L_C -path.

Finally, a variable v points to an object O iff there exists an L_{FC} -path p from O to v , such that $L_F(p) \in L_F$ (p is a *flowsto-path*) and $L_C(p) \in L_C$ (p is a *realizable-path*). Ignoring all balanced contexts, the contexts for v and O can be directly read off from p (Section 3.2.2).

 **Table 1** Points-to results for the program in Figure 3 computed by 2CFA according to Figure 1.

Method	Pointers	PTS	Method	Pointers	PTS
main()	$\langle o_1, [] \rangle$	$\{\langle O1, [] \rangle\}$	bar()	$\langle x, [c1] \rangle$	$\{\langle A1, [] \rangle\}$
	$\langle o_2, [] \rangle$	$\{\langle O2, [] \rangle\}$		$\langle o, [c1] \rangle$	$\{\langle O1, [] \rangle\}$
	$\langle a, [] \rangle$	$\{\langle A1, [] \rangle\}$		$\langle d, [c1] \rangle$	$\{\langle D1, [c1] \rangle\}$
	$\langle b, [] \rangle$	$\{\langle B1, [] \rangle\}$		$\langle x, [c2] \rangle$	$\{\langle B1, [] \rangle\}$
B:foo()	$\langle this, [c3, c2] \rangle$	$\{\langle B1, [] \rangle\}$		$\langle o, [c2] \rangle$	$\{\langle O2, [] \rangle\}$
	$\langle q, [c3, c2] \rangle$	$\{\langle D1, [c2] \rangle\}$		$\langle d, [c2] \rangle$	$\{\langle D1, [c2] \rangle\}$
A:foo()	$\langle this, [c3, c1] \rangle$	$\{\langle A1, [] \rangle\}$	Field	Pointers	PTS
	$\langle p, [c3, c1] \rangle$	$\{\langle D1, [c1] \rangle\}$	f	$\langle D1.f, [c1] \rangle$	$\{\langle O1, [] \rangle\}$
	$\langle v, [c3, c1] \rangle$	$\{\langle O1, [] \rangle\}$		$\langle D1.f, [c2] \rangle$	$\{\langle O2, [] \rangle\}$

2.2 Motivation

We begin with a motivating example (Section 2.2.1) and an inclusion-based framework featuring on-the-fly call graph construction (Section 2.2.2). We explore the limitations of L_{FC} without this feature (Section 2.2.3) and the challenges of developing L_{DCR} with it (Section 2.2.4). Transitioning from L_{FC} to L_{DCR} requires a new PAG representation specific to L_{DCR} .

2.2.1 Example

In Figure 3, classes A, B, C, D, and O are defined. B and C, subclasses of A, override the `foo()` method from A. The notation `T:m()` represents method `m()` in class `T`. The method `bar()` is a wrapper, storing the object pointed to by `o` in `D1.f`, and then invoking `A:foo()`, `B:foo()`, or `C:foo()` based on the dynamic type of object `x` points to. In `main()`, `O1`, `O2`, `A1`, and `B1` are created, in which `A1` and `O1` (`B1` and `O2`) are passed into `bar()` as its two arguments at callsite `c1` (`c2`).

2.2.2 Inclusion-based Formulation

Table 1 lists the points-to results computed for the program in Figure 3 by 2CFA following the rules in Figure 1. For `main()`, analyzed under `[]`, its points-to relations are obtained trivially. As for `bar()`, there are two calling contexts, `[c1]` and `[c2]`. Under `[c1]`, we have $PTS(x, [c1]) = \{\langle A1, [] \rangle\}$, $PTS(d, [c1]) = \{\langle D1, [c1] \rangle\}$, and $PTS(D1.f, [c1]) = PTS(o, [c1]) = \{\langle O1, [] \rangle\}$. Then `A:foo()` is found to be the target invoked by `x.foo()` at callsite `c3` in line 17 ([I-VCALL]). Thus, $PTS(p, [c3, c1]) = \{\langle D1, [c1] \rangle\}$ and $PTS(v, [c3, c1]) = \{\langle O1, [] \rangle\}$. Similarly, when `bar()` is analyzed under `[c2]`, we have $PTS(x, [c2]) = \{\langle B1, [] \rangle\}$. Thus, `x.foo()` at callsite `c3` is now resolved to `B:foo()`. Note that [I-VCALL] supports on-the-fly call graph construction during the analysis and 2CFA is precise enough by not resolving `C:foo()` as a spurious target at `c3`.

2.2.3 L_{FC} -based Formulation

L_{FC} addresses k CFA using a separate call graph construction algorithm. This approach separates, both conceptually and algorithmically, the parameter passing at a virtual callsite from the dynamic dispatch process. The limitations arising from this separation are explored below, considering whether the call graph is pre-constructed or constructed on the fly.

In Figure 3, L_{FC} uses a PAG as shown in Figure 4, constructed with CHA [9], an imprecise yet fast and sound call graph algorithm. In this scenario, `C:foo()` is conservatively marked as a target method at callsite `c3` (line 17). However, as explained later, L_{FC} would exclude such spurious targets when employing a more precise call graph in its analysis.

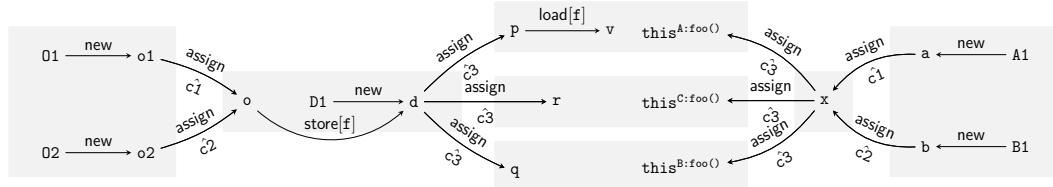


Figure 4 The PAG operated on by L_{FC} for the program given in Figure 3.

We analyze a specific traversal path leading to d , an argument in the call to $\text{foo}()$ at callsite c_3 (line 17), originating from o_1 in $\text{bar}(a, o_1)$ under $[c_1]$ or o_2 in $\text{bar}(b, o_2)$ under $[c_2]$. The subsequent task is to assign d to the appropriate parameter, based on the target method identified at this callsite: p for $A:\text{foo}()$, q for $B:\text{foo}()$, or r for $C:\text{foo}()$.

2.2.3.1 Using a Call Graph Constructed in Advance

Even if L_{FC} uses the most precise pre-built call graph, $k\text{CFA}$ can still lose precision. For instance, at callsite c_3 (line 17) in Figure 3, both $A:\text{foo}()$ and $B:\text{foo}()$ are identified as possible target methods. This means $A:\text{foo}()$ is always considered a target method, whether the call is from $\text{bar}(a, o_1)$ under $[c_1]$ or $\text{bar}(b, o_2)$ under $[c_2]$. As a result, this scenario leads to the identification of two L_{FC} -paths:

$$o_1 \xrightarrow{\text{new}} o_1 \xrightarrow[\hat{c}_1]{\text{assign}} o \xrightarrow{\text{store}[f]} d \xrightarrow{\text{new}} D_1 \xrightarrow{\text{new}} d \xrightarrow[\hat{c}_3]{\text{assign}} p \xrightarrow{\text{load}[f]} v \quad (3)$$

$$o_2 \xrightarrow{\text{new}} o_2 \xrightarrow[\hat{c}_2]{\text{assign}} o \xrightarrow{\text{store}[f]} d \xrightarrow{\text{new}} D_1 \xrightarrow{\text{new}} d \xrightarrow[\hat{c}_3]{\text{assign}} p \xrightarrow{\text{load}[f]} v \quad (4)$$

Thus, in this L_{FC} -based pointer analysis, v is concluded to point to both o_1 and o_2 , despite v actually pointing only to o_1 as per 2CFA (Table 1), meaning that o_2 is spurious.

L_{FC} 's precision loss stems from its approach to parameter passing at virtual callsites ([P-VCALL]), treating them similarly to static callsites ([P-SCALL]) using inter-procedural `assign` edges, without accounting for CFL-reachability for specific receiver objects. As a result, this causes L_{FC} to overlook that the L_{FC} -path in Equation (3) and the L_{FC} -path in Equation (4) are relevant only when x points to A_1 at $[c_1]$ and B_1 at $[c_2]$, respectively.

If L_{FC} uses a less precise call graph, which is pre-built by, say, CHA [9], then $C:\text{foo}()$ will also be identified as a target method at callsite c_3 (line 17), leading to r pointing to D_1 spuriously due to $D_1 \xrightarrow{\text{new}} d \xrightarrow[\hat{c}_3]{\text{assign}} r$. However, r 's points-to set is actually empty as per 2CFA (not listed in Table 1).

2.2.3.2 Using a Call Graph Constructed On the Fly

When d is reached at callsite c_3 in line 17 of Figure 3, using a call graph constructed on the fly as in demand-driven analyses [53, 62, 48], where methods invoked at a virtual callsite are context-specific, enables us to discern that the path in Equation (3) is an L_{FC} -path, while that in Equation (4) is not. This precision ensures that v points only to o_1 . In the first path, x points to A_1 under context $[c_1]$, identifying $A:\text{foo}()$ as the target at c_3 . The path $\xrightarrow[\hat{c}_3]{\text{assign}} p \xrightarrow{\text{load}[f]} v$ confirms that v points to o_1 . In the second path, reaching d under $[c_2]$ leads to $B:\text{foo}()$ at c_3 (with x pointing to B_1), blocking the same path.

While L_{FC} can address $k\text{CFA}$ on-demand more accurately than a pre-built call graph, precision loss may still occur in scenarios where a callsite has multiple dispatch targets under a common context. For example, in Figure 5 (where classes E, F, and G are renamed

```

1 class E {
2   void foo(G p) {
3     Object v = p.g;
4   }
5 class F extends E {
6   void foo(G q) { }
7 }
8 class G { Object g; }
9 G w = new G(); // G1
10 if (...) {
11   E e1 = new E(); // E1
12   w.g = e1;
13 } else {
14   F f1 = new F(); // F1
15   w.g = f1;
16 }
17 E x = w.g;
18 x.foo(null); // c

```

Figure 5 A small example.

from classes A, B, and D in Figure 3 to prevent name collisions), using a separate call graph construction algorithm to identify all potential target methods at “`x.foo(null)`” under any context results in the discovery of both `E:foo()` and `F:foo()`. Subsequent analysis of CFL-reachability with L_{FC} yields:

$$E1 \xrightarrow{\text{new}} e1 \xrightarrow{\text{store}[g]} w \xrightarrow{\text{new}} G1 \xrightarrow{\text{new}} w \xrightarrow{\text{load}[g]} x \xrightarrow[\hat{c}]{\text{assign}} \text{this}^{E:\text{foo}()} \quad (5)$$

$$F1 \xrightarrow{\text{new}} f1 \xrightarrow{\text{store}[g]} w \xrightarrow{\text{new}} G1 \xrightarrow{\text{new}} w \xrightarrow{\text{load}[g]} x \xrightarrow[\hat{c}]{\text{assign}} \text{this}^{E:\text{foo}()} \quad (6)$$

Therefore, both `E1` and `F1` will flow to `thisE:foo()` although `F1` is spurious by [I-VCALL]. Similarly, both `E1` and `F1` will flow to `thisF:foo()` with `E1` being spurious.

L_{FC} ’s precision loss stems from treating the receiver variable the same as other arguments ([P-VCALL] in Figure 2), in contrast to the inclusion-based approach ([I-VCALL] in Figure 1). Attempting to eliminate spurious receiver objects like `F1` for `E:foo()` informally, outside the specifications of L_{FC} or any call graph construction algorithm, is an ad hoc solution. This problem has persisted in the L_{FC} on-demand algorithm for kCFA [53], released as part of the SOOT compiler [59] and used by many other researchers [61, 48], in the last 15 years.

2.2.3.3 Discussion

In addressing kCFA, L_{FC} depends on an external algorithm for call graph construction. This approach not only leads to the precision loss in kCFA as previously mentioned, but also presents another limitation: L_{FC} is unable to track all value-flow paths involved in method dispatch, whether the call graph is constructed beforehand or generated on-the-fly.

In analyzing “`x.foo(d)`” in line 17 of Figure 3, for parameter passing of `d` at the callsite as per [I-VCALL], it is necessary to first identify methods dispatched on the receiver objects that `x` points to, then proceed with parameter passing (from `d` to `p` for `A:foo()`, and `d` to `q` for `B:foo()`). However, in L_{FC} , parameter passing, achieved through inter-procedural assign edges ([P-VCALL]), is conceptually and algorithmically detached from dynamic dispatch at the callsite. It does not relate this process via CFL-reachability to its receiver objects, a limitation also evident in the PAG shown in Figure 4.

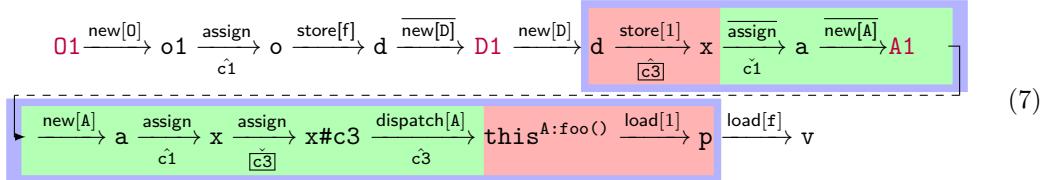
The limitations of L_{FC} indicate that its pre-analyses, designed to boost kCFA efficiency, can unintentionally compromise its precision. For example, SELECTX [33] aims to accelerate kCFA through selective context-sensitivity with L_{FC} , often leading to reduced precision.

2.2.4 L_{DCR} : Challenges and Our Solution

In developing L_{DCR} , it is crucial to facilitate CFL-reachability for parameter passing in line with k CFA. For a virtual call $\text{r.m}(a_1, \dots, a_n)$ at callsite c , passing an argument, denoted a , to its corresponding parameter p in a yet-to-be-discovered target method m' under context \mathbf{C} involves establishing a CFL-reachability path in a PAG representation, starting from a , through receiver variable r for dynamic dispatch (based on the dynamic type of the object pointed to by r under \mathbf{C}), and ending at p . Linking a to r , especially when $a \neq \text{r}$, is complex. Additionally, in CFL-reachability, some context elements in \mathbf{C} are consumed, i.e., matched during dynamic dispatch and must be restored for passing a to p under the same context \mathbf{C} . We identify three key challenges in handling this complex parameter-passing task:

- **CHL1.** How do we precisely pass r to the “this” variable of a target method m' invoked at callsite c , avoiding the precision loss as illustrated in Figure 5?
- **CHL2.** How do we establish a CFL-reachability path in a PAG representation of the program from a_i to p_i , passing through r to trigger dynamic dispatch during parameter passing, where p_i is the i -th parameter of a target method m' discovered at callsite c under \mathbf{C} ?
- **CHL3.** How do we ensure the passage of a_i to p_i for the target method m' invoked at callsite c with a context abstraction that accurately characterizes parameter passing for callsite c under \mathbf{C} ?

In our approach, illustrated using our motivating example (Figure 3), L_{DCR} is applied to a novel PAG representation depicted in Figure 7, distinct from the PAG used by L_{FC} (Figure 4). In this new formulation, we demonstrate that v points exclusively to 01 , attributable to a unique path from 01 to v :



The technical specifics of this path will be further elaborated in Section 3.

This path represents the flow of 01 to v through two calls, $c1$ (line 24) and $c3$ (line 17). Focusing on parameter passing of d at $c3$ under context $\mathbf{C} = [c1]$, where $A:\text{foo}()$ is the sole target, L_{DCR} employs a more indirect approach than L_{FC} 's direct inter-procedural assign edge $d \xrightarrow[c3]{\text{assign}} p$. L_{DCR} dynamically identifies dispatch targets in the path from d to p using a sequence of PAG edges. To address **CHL1**, we match $\text{new}[A]$ with $\text{dispatch}[A]$. For **CHL2**, d is stored in a special field of x to initiate dynamic dispatch, then loaded from the same field of $this^{A:\text{foo}()}$ into p (highlighted in \blacksquare). Afterwards, dynamic dispatch under $\mathbf{C} = [c1]$ is performed similarly to L_{FC} (highlighted in \blacksquare). To tackle **CHL3**, d is passed to p under context $[c3, c1]$, where $c3$ denotes the callsite and $c1$ the context for $A1$ flowing into x (highlighted in \blacksquare). The importance of the two boxed below-edge labels, $\hat{c3}$ and $\check{c3}$, in meeting **CHL3** will be elaborated upon in Section 3.

3 L_{DCR} : Design and Insights

When tackling a CFL-reachability problem, the selection of CFLs and their corresponding graph representations are closely interconnected and thoughtfully designed. To separate this interdependency, we first introduce a new PAG representation for a program, which supports

$$\begin{array}{c}
\frac{x = \mathbf{new} T // O}{O \xrightarrow{\mathbf{new}[T]} x} [\mathbf{C-NEW}] \quad \frac{M \text{ is an instance method}}{\mathbf{this}^M \xrightarrow{\mathbf{load}[i]} p_i^M} [\mathbf{C-PARAM}] \quad \frac{M \text{ is an instance method}}{\mathbf{ret}^M \xrightarrow{\mathbf{store}[0]} \mathbf{this}^M} [\mathbf{C-RET}] \\
\\
\frac{x = r.m(a_1, \dots, a_n) // c \quad t <: \text{DeclTypeOf}(r) \quad m' = \text{dispatch}(c, t)}{\forall i \in [1, n] : a_i \xrightarrow[\boxed{\hat{c}}]{\mathbf{store}[i]} r \quad r \xrightarrow[\boxed{\check{c}}]{\mathbf{load}[0]} x \quad r \xrightarrow{\mathbf{assign}} r\#c \quad r \xrightarrow[\boxed{\hat{c}}]{\mathbf{assign}} r\#c \quad r\#c \xrightarrow[\boxed{\check{c}}]{\mathbf{dispatch}[t]} \mathbf{this}^{m'}} [\mathbf{C-VCALL}]
\end{array}$$

Figure 6 Rules for building the PAG required by L_{DCR} . **[C-ASSIGN]**, **[C-LOAD]**, **[C-STORE]** and **[C-SCALL]** mirror those in Figure 2 and are excluded here to conserve space.

on-the-fly call graph construction (Section 3.1). Following this, we elaborate on L_{DCR} by detailing our solutions to the three challenges (**CHL1 – CHL3**) and providing insights into its design (Section 3.2).

3.1 Pointer Assignment Graph

For representing a program in L_{DCR} , we employ the rules specified in Figure 6 to construct a PAG. The inverse of a PAG edge $x \xrightarrow[c]{\ell} y$, implicitly defined, is $y \xrightarrow[\bar{c}]{\bar{\ell}} x$, mirroring the approach in L_{FC} (Section 2.1.2). However, our approach uniquely allows below-edge labels to be also either $\boxed{\hat{c}}$ or \check{c} , where $\boxed{\hat{c}} = \check{c}$ and $\check{c} = \boxed{\hat{c}}$, with c denoting a callsite. To initiate dynamic dispatch at a callsite c , edges with boxed below-edge labels symbolize a novel type of inter-procedural value-flow entering (indicated by $\boxed{\hat{c}}$) or exiting (marked by \check{c}) a method at c . These specific boxed below-edge labels are introduced solely for addressing **CHL3**, and their significance will be explained in Section 3.2.2.

Our PAG, designed for L_{DCR} , primarily differs from the one for L_{FC} (Figure 2) in handling virtual callsites. Consequently, **[C-ASSIGN]**, **[C-LOAD]**, **[C-STORE]**, and **[C-SCALL]** are the same as **[P-ASSIGN]**, **[P-LOAD]**, **[P-STORE]**, and **[P-SCALL]**, respectively. The additional rules in Figure 6 construct PAG edges that facilitate on-the-fly call graph construction at virtual callsites, addressing **CHL1** and **CHL2**.

In **[C-NEW]**, $O \xrightarrow{\mathbf{new}[T]} x$ specifically encodes T , the dynamic type of O , to facilitate dynamic dispatch on O and enable its use as a receiver object, avoiding precision loss as depicted in Figure 5.

For **[C-PARAM]** and **[C-RET]**, we treat the i -th (non-**this**) parameter of an instance method M (denoted as p_i^M , with i starting from 1) and its return variable \mathbf{ret}^M as special fields of \mathbf{this}^M , identified by offset i and 0, respectively. This allows the initialization of $\mathbf{this}^M.0$ with a store $\mathbf{ret}^M \xrightarrow{\mathbf{store}[0]} \mathbf{this}^M$ and a non-**this** parameter p_i^M with a load $\mathbf{this}^M \xrightarrow{\mathbf{load}[i]} p_i^M$.

In **[C-VCALL]**, we uniquely handle virtual calls like “ $x = r.m(a_1, \dots, a_n) // c$ ” differently from **[P-VCALL]** (Figure 2), using $r\#c$ to uniquely identify r at callsite c . There are two edges between r and $r\#c$: the edge $r \xrightarrow{\mathbf{assign}} r\#c$, which is essential for passing the receiver variable, and the edge $r \xrightarrow[\check{c}]{\mathbf{assign}} r\#c$, which is crucial for passing other arguments during parameter passing, as will be explained shortly. We initially over-approximate target methods at c using CHA ([9]), similar to L_{FC} , for later refinement by L_{DCR} . For each target method m' , the argument a_i is passed to the corresponding parameter $p_i^{m'}$ ($1 \leq i \leq n$) via a store $a_i \xrightarrow[\boxed{\hat{c}}]{\mathbf{store}[i]} r$ and a matching load $\mathbf{this}^{m'} \xrightarrow{\mathbf{load}[i]} p_i^{m'}$ (**[C-PARAM]**). CFL-reachability under L_{DCR} involves traversing this store edge to find the dynamic type of each receiver object pointed by r (marked by $\boxed{\hat{c}}$). The sequence $r \xrightarrow[\check{c}]{\mathbf{assign}} r\#c \xrightarrow[\check{c}]{\mathbf{dispatch}[t]} \mathbf{this}^{m'}$ indicates finding the dynamic type t (marked by \check{c}), enabling dispatch of m' with \hat{c} as its entry context (i.e.,

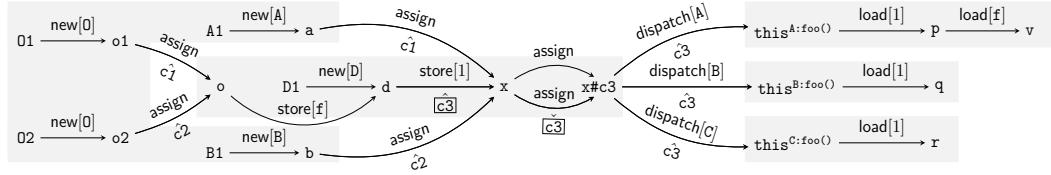


Figure 7 The PAG for L_{DCR} constructed for the program given in Figure 3.

$m' = \text{dispatch}(c, t)$ as desired). A **dispatch** edge also functions as an **assign** edge. For the receiver variable r , we simply use $r \xrightarrow{\text{assign}} r\#c$ (without the need for relating r to itself). Finally, x is assigned $\text{ret}^{m'}$ (stored in $\text{this}^{m'}.0$ ([**C-RET**])) via a load $r \xrightarrow{\text{load}[0]} x$, with \boxed{c} marking the conclusion of the dynamic dispatch at callsite c .

Figure 7 illustrates the PAG leveraged by L_{DCR} for our motivating example, as presented in Figure 3. This PAG, uniquely tailored to support L_{DCR} 's integrated call graph construction, shows notable differences from the PAG employed by L_{FC} , as depicted in Figure 4.

3.2 L_{DCR} : A New CFL-Reachability Formulation for k CFA

L_{DCR} combines three CFLs ($L_{DCR} = L_D \cap L_C \cap L_R$) for addressing **CHL1 – CHL3**. L_D , detailed in Section 3.2.1, deals with field accesses and dynamic dispatch, catering to **CHL1** and **CHL2**. L_C , defined in Equation (2), ensures callsite-sensitivity using below-edge labels Σ_{L_C} , which include \hat{c} and \check{c} , and treats L_{DCR} 's unique boxed labels $\hat{\square}$ and $\check{\square}$ as ϵ . L_R , presented in Section 3.2.2, facilitates parameter passing in on-the-fly call graph construction, addressing **CHL3**. The focus will predominantly be on L_D and L_R , concentrating on parameter passing, with method returns being similarly handled.

Basic Idea. L_{DCR} , a CFL-reachability formulation, differs from L_{FC} mainly in managing parameter passing at virtual callsites, enabling L_{DCR} 's built-in call graph construction compared to L_{FC} 's reliance on a separate algorithm (Sec. 2.2.3.2). At a virtual callsite “ $r.m(a_1, \dots, a_n); //c$ ” under context C , handling the receiver variable r (pointing to receiver objects) involves addressing **CHL1**: passing a receiver object to $\text{this}^{m'}$ for dispatch on m' . In addition, for an argument a_i , **CHL2** and **CHL3** are met by storing a_i in $r.i$, verifying if any object pointed by r under C matches dynamic type t , dynamically dispatching to m' ($m' = \text{dispatch}(c, t)$), and assigning $\text{this}^{m'}.i$ to $p_i^{m'}$ at callsite c under context C . Method returns are handled in a similar fashion.

► **Example 1.** Revisiting our motivating example (Figure 3) and its PAG (Figure 7), L_{DCR} ensures a unique path from **01** to v , as shown in Equation (7), so that v points only to **01** when $\text{bar}()$ is invoked at **c1**. The sub-path from **01** to d shows that **01** is stored into $d.f$, with d pointing to **D1**. The sub-path from d to p indicates parameter passing at callsite **c3** to p for $A:\text{foo}()$, dynamically identified by L_{DCR} under $C = [c1]$. We have discussed addressing **CHL1 – CHL3** at this callsite in Section 2.2.4. We wish to emphasize that $\hat{\square}$ and $\check{\square}$ signify dynamic dispatch's start and end at callsite **c3** for d . CFL-reachability traversal between these markers confirms that x points to **A1** under $[c1]$, necessitating a return to x under $[c1]$. With receiver object **A1**, $A:\text{foo}()$ is dispatched via $x\#c3 \xrightarrow[\hat{c3}]{\text{dispatch}[A]} \text{this}^{A:\text{foo}()}$, allowing d to pass to p under $[c3, c1]$. Unlike L_{FC} [53] that uses $[c3]$, L_{DCR} specifies $[c3, c1]$ to indicate this occurs only when x points to **A1** under $[c1]$. $C:\text{foo}()$, present in the PAG due to CHA [9], is filtered out by L_{DCR} 's on-the-fly call graph construction.

Let L_{FC}^{dd} be a demand-driven formulation of L_{FC} that is identical in all aspects except for one modification. This version continues to utilize a separate algorithm for on-the-fly call graph construction, but it has been specifically enhanced to accurately handle parameter passing for receiver variables, effectively avoiding the precision loss discussed in Section 2.2.3.2.

When developing L_{DCR} , we treat soundness fundamentally as an issue of precision.

► **Definition 2** (Soundness and Precision of On-the-Fly Call Graph Construction). *For any given callsite and context C , let T be the set of target methods identified under C through L_{FC}^{dd} . Suppose L is a language differing from L_{FC}^{dd} solely in managing parameter passing at virtual callsites. We regard L as sound if it facilitates parameter passing under C for at least the methods in T , and as precise (besides being sound) if it enables parameter passing under C for precisely the target methods in T .*

We write $L_{DC} = L_D \cap L_C$ as the intersection of L_D and L_C . A path p qualifies as an L_{DCR} -path if $L_D(p) \in L_D$, $L_C(p) \in L_C$, and $L_R(p) \in L_R$. An L_{DC} -path is defined similarly. As we will discuss further, L_{DC} is sound yet imprecise, whereas L_{DCR} is precise.

3.2.1 The L_D Language

This CFL captures both field-sensitive accesses, similar to L_F in Equation (1), and dynamic dispatch within its language framework:

$$\begin{aligned}
\text{flowsto} &\longrightarrow \text{new}[t] (\text{flows} \mid \text{dispatch}[t])^* \\
\text{flows} &\longrightarrow \text{assign} \mid \text{store}[f] \text{ alias } \text{load}[f] \\
\text{alias} &\longrightarrow \overline{\text{flowsto}} \text{ flowsto} \\
\overline{\text{flowsto}} &\longrightarrow (\text{dispatch}[t] \mid \overline{\text{flows}})^* \overline{\text{new}[t]} \\
\overline{\text{flows}} &\longrightarrow \overline{\text{assign}} \mid \overline{\text{load}[f]} \text{ alias } \overline{\text{store}[f]}
\end{aligned} \tag{8}$$

Here, Σ_{L_D} includes all above-edge labels in the program's PAG. L_D extends L_F from Equation (1) [54, 53] by retaining its balanced parentheses approach for field accesses and adding support for dynamic dispatch, which facilitates on-the-fly call graph construction. Next, we describe how L_D is specifically designed to address **CHL1** and **CHL2**.

3.2.1.1 CHL1

To address **CHL1** regarding parameter passing at a virtual callsite, it is crucial that a receiver object O , pointed to by its receiver variable, is only passed to the `this` variable of a method dispatchable on O . For instance, in `x.foo(null)` from Figure 5, where `x` might point to both `E1` and `F1`, L_{FC} might incorrectly pass both `E1` and `F1` to `thisE:foo()`, as shown in Equations (5) and (6), despite `F1` being spurious. Note that L_{FC}^{dd} , introduced just before Definition 2, was specifically conceptualized to mitigate such precision loss.

In L_D , we explicitly specify the dynamic types of objects in four terminals: `new[t]`, `new[\overline{t}]`, `dispatch[t]`, and `dispatch[\overline{t}]`. This modification alters the two L_{FC} -paths discussed in Equations (5) and (6) for Figure 5 as follows:

$$E1 \xrightarrow{\text{new}[E]} e1 \xrightarrow{\text{store}[g]} w \xrightarrow{\overline{\text{new}[G]}} G1 \xrightarrow{\text{new}[G]} w \xrightarrow{\text{load}[g]} x \xrightarrow{\text{assign}} x\#c \xrightarrow[\epsilon]{\text{dispatch}[E]} \text{this}^{E:\text{foo}()} \tag{9}$$

$$F1 \xrightarrow{\text{new}[F]} f1 \xrightarrow{\text{store}[g]} w \xrightarrow{\overline{\text{new}[G]}} G1 \xrightarrow{\text{new}[G]} w \xrightarrow{\text{load}[g]} x \xrightarrow{\text{assign}} x\#c \xrightarrow[\epsilon]{\text{dispatch}[E]} \text{this}^{E:\text{foo}()} \tag{10}$$

During a `flowsto` ($\overline{\text{flowsto}}$) traversal, the type in `dispatch[t]` ($\overline{\text{dispatch}[t]}$) must align with its corresponding `new[t]` ($\overline{\text{new}[t]}$). Thus, the path in Equation (9) qualifies as an L_D -path, as $\text{new}[E] \text{ flows}^* \text{ dispatch}[E] \in L_D$, but the path in Equation (10) does not as $\text{new}[F] \text{ flows}^* \text{ dispatch}[E] \notin L_D$. Hence, in L_D , **F1** cannot spuriously flow to `thisE:foo()`. Similarly, in Equation (7), only **A1** can be passed to `thisA:foo()`, as `A:foo()` is dispatchable on **A1**.

► **Lemma 3.** *Consider a virtual callsite $x = r.m(a_1, \dots, a_n)$. In L_D , every receiver object pointed to by r flows only to the `this` variable of a method that can be dispatched on the receiver object.*

Proof Sketch. Follows from the definition of L_D . ◀

3.2.1.2 CHL2

To meet **CHL2** and trigger dynamic dispatch at virtual callsites during parameter passing, we use $L_{DC} = L_D \cap L_C$. Re-examining the L_{DCR} -path in Equation (7) without $\hat{c3}$ and $\check{c3}$, we assess if **01** flows into v starting from $c1$. Parameter passing for d at “`x.foo(d); // c3`” under $C = [c1]$ involves traversing the sub-path from d to p of `A:foo()`. Starting with $d \xrightarrow{\text{store}[1]} x$, a `flowsto` traversal is initiated via $x \xrightarrow[\check{c1}]{\text{assign}} a \xrightarrow{\text{new}[A]} \text{A1}$ under $C = [c1]$, returning to x via $\text{A1} \xrightarrow{\text{new}[A]} a \xrightarrow[\check{c1}]{\text{assign}} x$, dispatching at $c3$ through $x \xrightarrow{\text{assign}} x\#c3 \xrightarrow[\hat{c3}]{\text{dispatch}[A]} \text{this}^{A:\text{foo}()}$, and finally passing d to p via $\text{this}^{A:\text{foo}()} \xrightarrow{\text{load}[1]} p$. Unlike L_{FC} 's direct passage of d to p in Equation (3), L_{DCR} uses a series of edges under $[c3, c1]$, indicating dispatch occurs only when x points to **A1** under $[c1]$.

► **Lemma 4.** *L_{DC} is sound in handling parameter passing at virtual callsites.*

Proof Sketch. Consider a virtual callsite $r.m(a_1, \dots, a_n); // c$, where parameter passing for an argument occurs under context C . Let T represent the set of target methods identified on the fly for this callsite under C by applying a separate call graph algorithm as in L_{FC}^{dd} . As r is handled similarly as in L_{FC}^{dd} , it suffices to consider parameter passing for a non-this argument a_i . Focusing on a_i , L_{DC} initiates dynamic dispatch by locating receiver objects pointed to by r under also C . Since L_{DC} differs from L_{FC}^{dd} only in handling parameter passing at virtual callsites, the set of target methods found by L_{DC} must include T . In addition, for each target $m' \in T$, there always exists a PAG path q :

$$a_i \xrightarrow{\text{store}[i]} r \xrightarrow{\text{flowsto}} O \xrightarrow{\text{flowsto}} r \xrightarrow{\text{assign}} r\#c \xrightarrow{\text{dispatch}[-]} \text{this}^{m'} \xrightarrow{\text{load}[i]} p_i^{m'} \quad (11)$$

Here, if u represents “ $r \xrightarrow{\text{flowsto}} O$ ”, then “ $O \xrightarrow{\text{flowsto}} r$ ” is its inverse \bar{u} . This ensures a_i flows p_i by L_D and $L_C(q) \in L_C$ by L_C . Moreover, $L_C(q)$ forms a sequence of contexts feasible under C , as u is traversed under C . Therefore, by Definition 2, L_{DC} is sound. ◀

3.2.2 The L_R Language

L_{DC} , though sound, is not precise. This is illustrated in examples from Figures 8 and 9, highlighting L_{DC} 's limitations and underscoring the importance of L_R in L_{DCR} .

```

1 static void main() {
2   H h = new H(); // H1
3   I i1 = new I(); // I1
4   I i2 = new I(); // I2
5   h.m(i1); // c4
6   h.n(i2); // c5
7 }
8 class I {}
9 class H {
10  void m(Object p) { ... }
11  void n(Object q) { ... }
12 }

```

Figure 8 An example for illustrating the imprecision of L_{DC} caused by an incorrect dispatch site.

```

1 static void main() {
2   J j1 = new J(); // J1
3   K k1 = new K(); // K1
4   K k2 = new K(); // K2
5   K v1 = wid(j1, k1); // c6
6   K v2 = wid(j1, k2); // c7
7 }
8 class K {}
9 class J {
10  K id(K p) {
11    return p;
12 }
13 static K wid(J j, K k) {
14  K v = j.id(k); // c8
15  return v;
16 }

```

Figure 9 An example for showing the imprecision of L_{DC} caused by an incorrect dispatch context.

L_{DC} 's precision loss can occur from a spurious dispatch callsite, shown by the following two L_{DC} -paths for Figure 8, temporarily ignoring the boxed labels $\hat{c4}$, $\check{c4}$, and $\check{c5}$:

$$\text{I1} \xrightarrow{\text{new[I]}} \text{i1} \xrightarrow[\hat{c4}]{\text{store[1]}} \text{h} \xrightarrow{\text{new[H]}} \text{H1} \xrightarrow{\text{new[H]}} \text{h} \xrightarrow[\check{c4}]{\text{assign}} \text{h}\#c4 \xrightarrow{\text{dispatch[H]}} \text{this}^m \xrightarrow{\text{load[1]}} \text{p} \quad (12)$$

$$\text{I1} \xrightarrow{\text{new[I]}} \text{i1} \xrightarrow[\hat{c4}]{\text{store[1]}} \text{h} \xrightarrow{\text{new[H]}} \text{H1} \xrightarrow{\text{new[H]}} \text{h} \xrightarrow[\check{c5}]{\text{assign}} \text{h}\#c5 \xrightarrow{\text{dispatch[H]}} \text{this}^n \xrightarrow{\text{load[1]}} \text{q} \quad (13)$$

Both L_{DC} -paths track **I1**'s flow in the program's PAG. The first path correctly leads **I1** to **p**. However, the second path spuriously directs **I1** to **q**, as the flowsto traversal to identify **a**'s receiver object starts at **c4** but concludes at **c5** spuriously. L_R addresses this precision issue by requiring matched boxed edge labels. As a result, the first path in Equation (12) is a valid L_{DCR} -path (with $\hat{c4}$ matched by $\check{c4}$), while the second path in Equation (13) is invalidated (due to the mismatch of $\hat{c4}$ and $\check{c5}$).

L_{DC} may also experience precision loss due to a spurious dispatch context. Consider the following two L_{DC} -paths in the PAG of Figure 9 (by ignoring the boxed labels $\hat{c8}$ and $\check{c8}$ for now):

$$\begin{aligned} & \text{K1} \xrightarrow{\text{new[K]}} \text{k1} \xrightarrow[\hat{c6}]{\text{assign}} \text{k} \xrightarrow{\text{store[1]}} \text{j} \xrightarrow[\check{c6}]{\text{assign}} \text{j1} \xrightarrow{\text{new[J]}} \text{J1} \xrightarrow{\text{new[J]}} \text{j1} \xrightarrow[\hat{c6}]{\text{assign}} \text{j} \xrightarrow[\check{c8}]{\text{assign}} \text{j}\#c8 \xrightarrow{\text{dispatch[J]}} \text{this}^{\text{id}} \xrightarrow{\text{load[1]}} \\ & \text{p} \xrightarrow{\text{store[0]}} \text{this}^{\text{id}} \xrightarrow[\hat{c8}]{\text{dispatch[J]}} \text{j}\#c8 \xrightarrow[\check{c8}]{\text{assign}} \text{j} \xrightarrow[\hat{c6}]{\text{assign}} \text{j1} \xrightarrow{\text{new[J]}} \text{J1} \xrightarrow{\text{new[J]}} \text{j1} \xrightarrow[\hat{c6}]{\text{assign}} \text{j} \xrightarrow[\check{c8}]{\text{load[0]}} \text{v} \xrightarrow[\hat{c6}]{\text{assign}} \text{v1} \end{aligned} \quad (14)$$

$$\begin{aligned} & \text{K1} \xrightarrow{\text{new[K]}} \text{k1} \xrightarrow[\hat{c6}]{\text{assign}} \text{k} \xrightarrow{\text{store[1]}} \text{j} \xrightarrow[\check{c6}]{\text{assign}} \text{j1} \xrightarrow{\text{new[J]}} \text{J1} \xrightarrow{\text{new[J]}} \text{j1} \xrightarrow[\hat{c7}]{\text{assign}} \text{j} \xrightarrow[\check{c8}]{\text{assign}} \text{j}\#c8 \xrightarrow{\text{dispatch[J]}} \text{this}^{\text{id}} \xrightarrow{\text{load[1]}} \\ & \text{p} \xrightarrow{\text{store[0]}} \text{this}^{\text{id}} \xrightarrow[\hat{c8}]{\text{dispatch[J]}} \text{j}\#c8 \xrightarrow[\check{c8}]{\text{assign}} \text{j} \xrightarrow[\hat{c7}]{\text{assign}} \text{j1} \xrightarrow{\text{new[J]}} \text{J1} \xrightarrow{\text{new[J]}} \text{j1} \xrightarrow[\hat{c7}]{\text{assign}} \text{j} \xrightarrow[\check{c8}]{\text{load[0]}} \text{v} \xrightarrow[\hat{c7}]{\text{assign}} \text{v2} \end{aligned} \quad (15)$$

These two L_{DC} -paths in Figure 9 vary only by context: Equation (15) is similar to Equation (14), but replaces **c7** with **c6** and **v2** with **v1**. Both track where **K1** flows, starting from “**wid(j1,k1); // c6**”. According to Equation (14), **v1** points to **K1** as expected. However, Equation (15) inaccurately allows **K1**, passed at **c6**, to flow into **v2** at **c7**, spuriously indicating

v2 points to K1. Focusing on dynamic dispatch at callsite c8 in line 14 due to the call at c6 in line 5 (Figure 9), Equation (14) shows that j initially pointing to J1 under [c6] and maintains this during both flowsto and flowsto traversals from c6. However, Equation (15) starts similarly but ends with j pointing to J1 under [c7], which is inconsistent with the call at c6.

In general, L_{DC} may lack precision as it sometimes includes spurious sub-paths for dynamic dispatch. Consider a generic virtual callsite $r.m(a_1, \dots, a_n) // c$, L_{DC} initiates dynamic dispatch by executing the following alias-related traversal on its receiver variable r:

$$\dots \xrightarrow[\hat{c}]{\text{store}[i]} r \xrightarrow{\text{flowsto}} O \xrightarrow{\text{flowsto}} r' \xrightarrow[\check{c}']{\text{assign}} r' \# c' \xrightarrow{\hat{c}'} \dots \quad (16)$$

Such a *dispatch path*, which starts from \hat{c} and ends at \check{c}' , is *valid* if two conditions are met:

- **DP-C1:** $c = c'$ (implying that $r = r'$), and
- **DP-C2:** O is pointed by both r and r' under exactly the same context.

However, L_{DC} can ensure that r and r' are aliases but cannot guarantee the validity of this dispatch path. For example, Equation (13) contains a dispatch path violating **DP-C1**, and Equation (15) violates **DP-C2**. To exclude such invalid dispatch paths in L_{DC} -paths, L_R is designed to utilize all below-edge labels in the PAG (i.e., \hat{c} , \check{c} , \hat{C} , and \check{C}) as terminals:

$$\begin{aligned} \text{recoveredCtx} &\longrightarrow \text{recoveredCtx } \hat{c} \mid \text{recoveredCtx } \check{c} \mid \text{recoveredCtx siteRecovered} \mid \epsilon \\ \text{siteRecovered} &\longrightarrow \hat{C} \text{ ctxRecovered } \check{C} \\ \text{ctxRecovered} &\longrightarrow \text{matched ctxRecovered} \mid \text{ctxRecovered matched} \mid \check{c} \text{ ctxRecovered } \hat{c} \mid \epsilon \\ \text{matched} &\longrightarrow \text{matched matched} \mid \hat{c} \text{ matched } \check{c} \mid \text{siteRecovered} \mid \epsilon \end{aligned} \quad (17)$$

Here, Σ_{L_R} includes all below-edge labels in the program's PAG. The start symbol `recoveredCtx` would define a language that contains L_C if its third alternative “`recoveredCtx siteRecovered`” were changed to “`recoveredCtx`”. Thus, L_R is engaged during a dispatch path traversal. The `siteRecovered` production enforces **DP-C1**, and the `ctxRecovered` and `matched` productions collectively enforce **DP-C2**. This design enables L_R to address **CHL3** by reinstating the context of r.

By incorporating L_R into L_{DC} , the composite language $L_{DCR} = L_D \cap L_C \cap L_R$ achieves precision in managing parameter passing at virtual callsites. Reexamining the paths in Equations (14) and (15), with the inclusion of \hat{C} and \check{C} , it is clear that the first path qualifies as an L_{DCR} -path, while the second does not. In the first path, the dynamic dispatch starts at callsite c8 under context [c6] and returns to the same callsite under the same context, signified by \hat{C} and \check{C} . Conversely, the second path, while also starting dispatch at callsite c8 under context [c6], mistakenly returns under a different context, [c7], making it invalid for L_{DCR} . As a result, L_{DCR} correctly determines that K1 is pointed to by v1, but not by v2, effectively preventing v2 from pointing to K1 spuriously.

Below, we give a formal development of L_R , followed by a proof of L_{DCR} 's precision.

To determine the points-to set of a variable v , $\text{PTS}(v, c_v)$, using L_{DC} , consider an L_C -path p with label $L_C(p) = \ell_1, \dots, \ell_n$, where each ℓ_i is a context label on an inter-procedural assign edge. The inverse of p , \bar{p} , has a label $L_C(\bar{p}) = \bar{\ell}_n, \dots, \bar{\ell}_1$. Splitting p into sub-paths p^{ex} and p^{en} , we define $L_C^{\text{ex}}(p) = L_C(p^{\text{ex}})$ and $L_C^{\text{en}}(p) = L_C(p^{\text{en}})$, with $L_C(p) = L_C^{\text{ex}}(p)L_C^{\text{en}}(p)$. Here, $L_C^{\text{ex}}(p)$ and $L_C^{\text{en}}(p)$ are derived from exit and entry in L_C 's grammar (Equation (2)). For $s \in L_C$, $\mathcal{B}(s)$ returns s's canonical form with balanced contexts removed. If c is a string of exit contexts like $\check{c}_1 \dots \check{c}_n$, $\mathcal{E}(c) = [c_1, \dots, c_n]$ converts it into a context representation, noting $\mathcal{E}(\epsilon) = []$.

For an L_{DC} -path p from an object O to a variable v , we can clearly deduce the following points-to relationship, including the specific contexts of O and v :

$$\langle O, \mathcal{E}(\mathcal{B}(L_C^{\text{ex}}(p))) \rangle \in \text{PTS}(v, \mathcal{E}(\overline{\mathcal{B}(L_C^{\text{en}}(p))})) \quad (18)$$

► **Example 5.** Let us take $p_{\text{O1},v}$, the L_{DC} -path from Equation (7), by ignoring $\hat{[\text{c3}]}$ and $\check{[\text{c3}]}$. By definition, $L_C(p_{\text{O1},v}) = \hat{[\text{c1c1c1c3}]}$, where $p_{\text{O1},v}^{\text{ex}}$ denotes the sub-path from O1 to A1 and $p_{\text{O1},v}^{\text{en}}$ denotes the sub-path from A1 to v . Thus, $L_C^{\text{ex}}(p_{\text{O1},v}) = \hat{[\text{c1c1}]}$ and $L_C^{\text{en}}(p_{\text{O1},v}) = \hat{[\text{c1c3}]}$. Since $\mathcal{E}(\mathcal{B}(\hat{[\text{c1c1}]}) = []$ and $\mathcal{E}(\mathcal{B}(\hat{[\text{c1c3}]}) = [\text{c3}, \text{c1}]$, we have: $\langle \text{O1}, [] \rangle \in \text{PTS}(v, [\text{c3}, \text{c1}])$.

To enforce **DP-C1**, the production $\text{siteRecovered} \rightarrow \hat{[\text{c}]} \text{ ctxRecovered } \check{[\text{c}]}$ ensures that a dispatch process starting at a callsite (indicated by $\hat{[\text{c}]}$) concludes at the same callsite (marked by $\check{[\text{c}]}$). In the dispatch path from Equation (16), this guarantees $c = c'$ and $r = r'$. Thus, matching $\hat{[\text{c}]}$ with $\check{[\text{c}]}$ allows c to be reinstated at the next dispatch edge, ensuring dynamic dispatch occurs specifically at callsite c .

To enforce **DP-C2**, the ctxRecovered - and matched -productions are crucial, with ctxRecovered

$\rightarrow \check{[\text{c}]} \text{ ctxRecovered } \hat{[\text{c}]}$ being central. This is best understood through a generic dispatch path in Equation (16). **DP-C2** can be rephrased as follows. Let $p_{r,O}$ be the flowsto path from r to O , and its inverse $\overline{p_{r,O}}$ a flowsto path. Consider $p_{O,r'}$ as the flowsto path from O to r' . The path from r to r' is composed of $p_{r,O} p_{O,r'}$ or equivalently $p_{r,O}^{\text{ex}} p_{r,O}^{\text{en}} p_{O,r'}^{\text{ex}} p_{O,r'}^{\text{en}}$. Applying Equation (18), we deduce:

$$\begin{aligned} \langle O, \mathcal{E}(\mathcal{B}(L_C^{\text{ex}}(\overline{p_{r,O}}))) \rangle &\in \text{PTS}(r, \mathcal{E}(\overline{\mathcal{B}(L_C^{\text{en}}(\overline{p_{r,O}}))})) \\ \langle O, \mathcal{E}(\mathcal{B}(L_C^{\text{ex}}(p_{O,r'}))) \rangle &\in \text{PTS}(r', \mathcal{E}(\overline{\mathcal{B}(L_C^{\text{en}}(p_{O,r'}))})) \end{aligned} \quad (19)$$

As r and r' are aliases, they must always point to O with exactly the same heap context, i.e., $\mathcal{E}(\mathcal{B}(L_C^{\text{ex}}(\overline{p_{r,O}}))) = \mathcal{E}(\mathcal{B}(L_C^{\text{ex}}(p_{O,r'})))$. Thus, $\mathcal{B}(\mathcal{B}(L_C^{\text{ex}}(\overline{p_{r,O}})) \mathcal{B}(L_C^{\text{ex}}(p_{O,r'}))) = \epsilon$ holds, implying the edge labels on path $p_{r,O}^{\text{en}} p_{O,r'}^{\text{ex}}$ must be balanced. Besides, r and r' are required to have the same context, i.e., $\mathcal{E}(\mathcal{B}(L_C^{\text{en}}(\overline{p_{r,O}}))) = \mathcal{E}(\mathcal{B}(L_C^{\text{en}}(p_{O,r'})))$. Thus, the following must be true:

$$\mathcal{B}(\mathcal{B}(L_C^{\text{en}}(p_{O,r'})) \overline{\mathcal{B}(L_C^{\text{en}}(\overline{p_{r,O}}))}) = \epsilon \quad (20)$$

implying that the edge labels in path $p_{O,r'}^{\text{en}} p_{r,O}^{\text{ex}}$ must be balanced out.

Both the ctxRecovered - and matched -productions in L_R play key roles during dispatch path traversal, as illustrated in Equation (16). The production $\text{ctxRecovered} \rightarrow \check{[\text{c}]} \text{ ctxRecovered } \hat{[\text{c}]}$ enforces **DP-C2** (see Equation (20)), while $\text{matched} \rightarrow \text{siteRecovered}$ initiates traversal of another dispatch path. The other productions help bypass matched contexts and callsites. In simple terms, for a traversal from r to O (r flowsto O), writing down all unmatched exit contexts as $\check{[\text{c}_1]}, \dots, \check{[\text{c}_n]}$ implies that the unmatched entry contexts seen on the return from O to r' (O flowsto r') should be $\hat{[\text{c}_1]}, \dots, \hat{[\text{c}_n]}$.

Revisiting the two L_{DC} -paths from Equations (14) and (15), as introduced in Section 3.2.2, the L_{DC} -path in Equation (14) qualifies as an L_{DCR} -path due to its valid dispatch paths. However, the L_{DC} -path in Equation (15) does not, as its initial dispatch path at callsite c8 from j to $j\#\text{c8}$ is invalid. With $\mathcal{B}(L_C^{\text{en}}(p_{j,\text{J1}})) = \check{[\text{c6}]}$ and $\mathcal{B}(L_C^{\text{en}}(p_{\text{J1},j})) = \hat{[\text{c7}]}$, we find $\mathcal{B}(\mathcal{B}(L_C^{\text{en}}(p_{\text{J1},j})) \mathcal{B}(L_C^{\text{en}}(p_{j,\text{J1}}))) = \hat{[\text{c7}]} \check{[\text{c6}]} \neq \epsilon$, indicating the path is invalid as $\check{[\text{c6}]} \hat{[\text{c7}]}$ does not balance out according to $\text{ctxRecovered} \rightarrow \check{[\text{c}]} \text{ ctxRecovered } \hat{[\text{c}]}$.

► **Theorem 1.** L_{DCR} is precise in handling parameter passing for virtual callsites.

Proof. Drawing from Lemmas 3 and 4, it suffices to show that for every virtual callsite “ $r.m(a_1, \dots, a_n); // c$ ” under context \mathbf{C} , L_{DCR} precisely handles parameter passing for the same target method set T identified at this callsite under \mathbf{C} by L_{FC}^{dd} ’s call graph algorithm. This holds as L_R filters out only those L_{DC} -paths with invalid dispatch paths. \blacktriangleleft

L_{DCR} achieves the same level of precision as L_{FC}^{dd} , thereby ensuring both soundness and precision in computing points-to information. We now employ L_{DCR} to determine points-to information in our motivating example (Figure 3), with Equation (18) being relevant but focusing solely on L_{DCR} -paths in the program’s PAG. Although CHA [9] in the PAG (Figure 7) broadly predicts target methods at virtual callsites, L_{DCR} ’s on-the-fly call graph construction process efficiently filters out spurious target methods like $\mathbf{C}:foo()$.

Finally, let us compare L_{DCR} , a CFL-reachability-based pointer analysis, with k CFA (Figure 1). While L_{DCR} , like L_{FC} [53], is suited for demand-driven analysis, k CFA is for whole-program analysis. Their key difference is the starting point: k CFA begins with entry methods M , including `main()`, and L_{DCR} with query variables V . Thus, k CFA may not compute points-to information for variables in V not reachable from M . In terms of precision, if k CFA determines $PTS(v, c)$ for variable v from M under context c , L_{DCR} can obtain exactly the same points-to set for v under c according to Equation (18). However, k CFA may overlook points-to information in the code unreachable from M .

3.3 Time Complexities

The PAG construction shown in Figures 2 and 6 scales linearly with the number of program statements. Yet, the L_{DCR} -reachability problem, like the L_{FC} -reachability problem [53], is undecidable due to being an intersection of three interwoven CFLs (L_D , L_C , L_R), making the combinations of $L_D \cap L_C$, $L_D \cap L_R$, and $L_C \cap L_R$ also undecidable [45]. For any individual CFL language $L \in \{L_D, L_C, L_R\}$, the reachability problem’s time complexity can reach up to $O(m^3n^3)$, where m is the grammar size and n is the number of nodes in the PAG.

Similar to k CFA (Figure 1), which introduces k -limiting to L_C in L_{FC} , resulting in a complexity of $O(n^3)$, we can also render the L_{DCR} -reachability problem computable within polynomial time for practical applications by applying k -limiting to both L_C and L_R .

4 P3Ctx : An Application of L_{DCR}

In our secondary contribution, we demonstrate the effectiveness of L_{DCR} through P3Ctx, the first pre-analysis tool powered by L_{DCR} for accelerating k CFA with selective context-sensitivity, always maintaining its precision. This also confirms L_{DCR} ’s correctness. Conversely, SELECTX [33], an L_{FC} -enabled pre-analysis does not guarantee precision preservation.

4.1 Selective Context-Sensitivity

Selective context sensitivity enhances the efficiency of context-sensitive analyses, maintaining much of their precision. It applies context-sensitivity selectively to crucial program variables and objects, treating the rest context-insensitively. SELECTX [33], a recent method for selective context-sensitive pointer analysis in k CFA, is built on L_{FC} , an incomplete formulation dependent on an external call graph construction algorithm. As a result, SELECTX inaccurately categorizes some vital variables and objects, causing precision loss. To remedy this, we introduce P3Ctx, a new L_{DCR} -based pre-analysis technique for selective context-sensitivity in k CFA, ensuring precision. P3Ctx is developed following the fundamental approach used in [33] for creating SELECTX.

4.1.1 CFL-Reachability-Guided Selections

Applying L_{FC} to develop SELECTX [33] is straightforward. For a flowsto path $p_{O,n,v}$ in L_{FC} , starting from an object O to a variable v via n (a variable or object in method M), consider $p_{O,n}$ as the segment from O to n , and $p_{n,v}$ from n to v . Then n requires context-sensitivity in $kCFA$ to avoid potential precision loss *only if* three conditions are met:

$$\begin{aligned} \text{CS-C1} : L_F(p_{O,n,v}) &\in L_F \\ \text{CS-C2} : L_C(p_{O,n}) &\in L_C \wedge L_C(p_{n,v}) \in L_C \\ \text{CS-C3} : L_C^{\text{en}}(p_{O,n}) &\neq \epsilon \wedge L_C^{\text{ex}}(p_{n,v}) \neq \epsilon \end{aligned} \quad (21)$$

where L_C^{en} and L_C^{ex} are from Section 3.2.2. O from outside M flows into n along $p_{O,n}$ context-sensitively and n flows out of M into v along $p_{n,v}$ context-sensitively, via M 's parameters (or return variable) along each path. Note that $p_{O,n,v}$ itself is not required to be an L_{FC} -path.

By replacing L_F with L_D in Equation (21), P3CTX also determines n to be context-sensitive *if* CS-C1–CS-C3 are met. Viewing these conditions as sufficient (rather than merely necessary) makes both SELECTX and P3CTX conservative, potentially marking some n as context-sensitive even when $kCFA$ would not lose precision with context-insensitive analysis. While SELECTX could lead to precision loss due to L_{FC} 's incompleteness, P3CTX, in contrast, always preserves precision. This is because L_{DCR} works with a PAG that clearly includes dispatch paths for all virtual callsites in the program.

► **Example 6.** In our motivating example (Figure 3), whether v spuriously points to D2 hinges on the context sensitivity of d , o , x , and D1 in $\text{bar}()$. Using L_{FC} and analyzing the PAG in Figure 4, SELECTX deems all four as context-insensitive, causing v to erroneously point to D2 because they cannot flow out of $\text{bar}()$ via its parameter x , failing to meet CS-C3. In L_{FC} 's PAG, which relies on an external call graph construction algorithm, there are no dispatch paths for these variables/objects to flow out of $\text{bar}()$ through x .

In L_{DCR} , the parameter passing of d at $x.\text{foo}(d)$ (line 17) directly relates to x via CFL-reachability (Figure 7). Consider $p_{\text{D1},n,v}$ in Equation (7), which is an L_{DCR} -path. For $n \in \{d, o, x, \text{D1}\}$, P3CTX designates each n as context-sensitive. This decision is because $p_{\text{D1},n,v}$ qualifies as an L_D -path (CS-C1), with both $p_{\text{D1},n}$ and $p_{n,v}$ being L_C -paths (CS-C2). Furthermore, $L_C^{\text{en}}(p_{\text{D1},n}) = \hat{c}1 \neq \epsilon$ and $L_C^{\text{ex}}(p_{n,v}) = \check{c}1 \neq \epsilon$, satisfying CS-C3.

4.1.2 Regularization

To make P3CTX as lightweight as possible so that we can efficiently make context-sensitivity selections without losing the performance benefits obtained from a subsequent main pointer analysis, we have decided to keep L_C unchanged as done in several earlier pre-analyses [35, 32, 33] but regularize L_D and L_R . We first regularize L_R to L_R^r as follows:

$$\text{recoveredCtx} \longrightarrow \text{recoveredCtx } \hat{c} \mid \text{recoveredCtx } \check{c} \mid \text{recoveredCtx } \hat{\square} \mid \text{recoveredCtx } \check{\square} \mid \epsilon \quad (22)$$

Thus, $L_D \cap L_C \cap L_R^r = L_D \cap L_C = L_{DC}$. By noting further that the boxed edge labels in L_R^r (i.e., $\hat{\square}$ and $\check{\square}$) are irrelevant to context-sensitivity selections and the regular entry/exit context labels in L_R^r (i.e., \hat{c} and \check{c}) have already been included in L_C , we conclude that L_R^r (i.e., L_R) can be ignored safely (or conservatively). As $L_{DC} \supseteq L_{DCR}$ (i.e., L_{DC} captures all the possible value-flows that are captured by L_{DCR} for a given program) according to Lemma 4, it suffices to use L_{DC} in place of L_{FC} in Equation (21) in developing our precision-preserving pre-analysis. Like the L_{FC} -reachability problem, the L_{DC} -reachability

problem is also undecidable [45]. Following [33], we regularize L_D into L_{D^r} and subsequently over-approximate L_{DC} to obtain $L_{D^r C} = L_{D^r} \cap L_C$. In Section 4.1.3, we present an algorithm to verify CS-C1–CS-C3 using $L_{D^r C}$ efficiently.

We start with $L_0 = L_D$. We first over-approximate L_0 by disregarding its field-sensitivity requirement and thus obtain L_1 given below:

$$\begin{array}{lcl} \text{flowsto} & \longrightarrow & \text{new } (\text{flows} \mid \text{dispatch})^* \\ \text{flows} & \longrightarrow & \text{assign} \mid \text{store } \overline{\text{flowsto}} \text{ flowsto load} \\ \overline{\text{flowsto}} & \longrightarrow & (\overline{\text{dispatch}} \mid \overline{\text{flows}})^* \text{ new} \\ \overline{\text{flows}} & \longrightarrow & \overline{\text{assign}} \mid \overline{\text{load}} \text{ flowsto store} \end{array} \quad (23)$$

In the absence of field-sensitivity, a `dispatch` ($\overline{\text{dispatch}}$) edge behaves just like an `assign` ($\overline{\text{assign}}$) edge and can thus be interpreted this way. As a result, we obtain L_2 below:

$$\begin{array}{lcl} \text{flowsto} & \longrightarrow & \text{new flows}^* \\ \overline{\text{flowsto}} & \longrightarrow & \overline{\text{flows}}^* \text{ new} \\ \text{flows} & \longrightarrow & \text{assign} \mid \text{store } \overline{\text{flowsto}} \text{ flowsto load} \\ \overline{\text{flows}} & \longrightarrow & \overline{\text{assign}} \mid \overline{\text{load}} \text{ flowsto store} \end{array} \quad (24)$$

Our approximation goes further by treating a `load` ($\overline{\text{load}}$) edge as also an `assign` ($\overline{\text{assign}}$). As a result, we will no longer require a `store` ($\overline{\text{load}}$) edge to be matched by a `load` ($\overline{\text{store}}$) edge. This will give rise to L_3 below:

$$\begin{array}{lcl} \text{flowsto} & \longrightarrow & \text{new flows}^* \\ \overline{\text{flowsto}} & \longrightarrow & \overline{\text{flows}}^* \text{ new} \\ \text{flows} & \longrightarrow & \text{assign} \mid \text{store } \overline{\text{flowsto}} \text{ flowsto} \\ \overline{\text{flows}} & \longrightarrow & \overline{\text{assign}} \mid \overline{\text{flowsto}} \text{ flowsto store} \end{array} \quad (25)$$

Finally, we obtain $L_{D^r} = L_4$ given below by no longer distinguishing a `store` edge from its inverse, `store` edge, so that we can represent both types of edges as a `store` edge:

$$\begin{array}{lcl} \text{flowsto} & \longrightarrow & \text{new flows}^* \\ \overline{\text{flowsto}} & \longrightarrow & \overline{\text{flows}}^* \text{ new} \\ \text{flows} & \longrightarrow & \text{assign} \mid \text{store } \overline{\text{assign}}^* \text{ new new} \\ \overline{\text{flows}} & \longrightarrow & \overline{\text{assign}} \mid \overline{\text{new}} \text{ new assign}^* \text{ store} \end{array} \quad (26)$$

► **Lemma 1.** $L_D \subseteq L_{D^r}$.

Proof. Follows from the fact that $L_i \subseteq L_{i+1}$. ◀

While L_{D^r} is identical to L_R regularized from L_F in SELECTX [33], our PAG (Figure 6), which makes dynamic dispatch paths explicitly, differs fundamentally from the one operated by L_{FC} (Figure 2). This distinction ensures that P3CTX preserves precision, unlike SELECTX.

Let $G = (N, E)$ be the PAG of a program. We use Andersen's algorithm [1] instead of CHA [9] to build its call graph in order to sharpen the precision of P3CTX.

We use a simple DFA shown in Figure 10 to accept L_{D^r} exactly. P3CTX runs inter-procedurally in linear time of the number of the PAG edges in G . To deal with L_C , we use summary edges added into the PAG (facilitated by the dotted transition labeled as `balanced`).

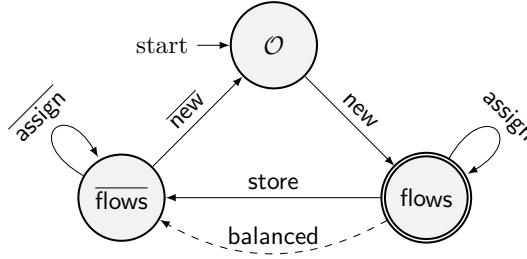


Figure 10 A DFA for accepting L_{D^r} .

4.1.3 P3Ctx

We follow [14] to develop a simple algorithm to verify **CS-C1–CS-C3** efficiently based on two properties that can be easily deduced from the DFA given in Figure 10 as stated below.

- Define $Q = \{\mathcal{O}, \text{flows}, \overline{\text{flows}}\}$ as the state set and $\delta : Q \times \Sigma \rightarrow Q$ as the transition function. For each PAG edge $n_1 \xrightarrow{\ell} n_2$ in G , the transition $\delta(q_1, \ell) = q_2$ leads to a one-step transition $(n_1, q_1) \rightarrow (n_2, q_2)$. The multiple-step transition \rightarrow^+ is the transitive closure of \rightarrow . The symmetry of flowsto and $\overline{\text{flowsto}}$ in L_{D^r} yields two straightforward properties of this DFA:
- **PROP-O.** Let O be an object created in a method M . Then $\langle \text{this}^M, \text{flows} \rangle \rightarrow^+ \langle O, \mathcal{O} \rangle \iff \langle O, \mathcal{O} \rangle \rightarrow^+ \langle \text{this}^M, \overline{\text{flows}} \rangle$ always holds.
 - **PROP-V.** Let v be a variable defined in a method M . Then $\langle \text{this}^M, \text{flows} \rangle \rightarrow^+ \langle v, q \rangle \iff \langle v, \overline{q} \rangle \rightarrow^+ \langle \text{this}^M, \overline{\text{flows}} \rangle$ always holds, where $q \in \{\text{flows}, \overline{\text{flows}}\}$ (since v is a variable).

To handle static callsites uniformly as virtual callsites, we assume that a static callsite is invoked on a dummy receiver object. Thus, in our PAG representation (Figure 6), passing arguments and receiving return values for a method must all flow through its “`this`” variable.

P3Ctx efficiently verify **CS-C1–CS-C3** as follows: For **CS-C1** (Equation (21)), where L_F is substituted with L_{D^r} , it is unnecessary to trace from an object along its flowsto paths. Instead, for each method, we start from its “`this`” variable, over-approximating that some object O can flow into it. For **CS-C2**, summary edges are utilized to confirm the balanced-parentheses property in L_C -paths. Finally, to ascertain **CS-C3**, we check for the existence of any $q \in Q$ such that:

$$\langle \text{this}^M, \text{flows} \rangle \rightarrow^+ \langle n, q \rangle \rightarrow^+ \langle \text{this}^M, \overline{\text{flows}} \rangle \quad (27)$$

where M is the containing method of n . This implies that n lies on an L_{D^r} -path collecting some values coming from outside M via `this`^M and pumping them out of M via `this`^M.

Let $R : Q \mapsto \wp(N)$ return the set of nodes in G reached at a state $q \in Q$. Then verifying **CS-C3**, i.e., checking Equation (27) involves determining if the following condition holds:

$$n \in R(\mathcal{O}) \quad \vee \quad n \in R(\text{flows}) \cap R(\overline{\text{flows}}) \quad (28)$$

Equation (27) is satisfied either when the first disjunct applies (due to **PROP-O**) or when the second disjunct applies (due to **PROP-V**).

Figure 11 outlines P3Ctx’s pre-analysis algorithm using three rules that streamline inter-procedural reachability in G . Here, $R^{-1} : N \mapsto \wp(Q)$ inversely maps nodes to their reachable states. The rules are: **[F-INIT]** for initializations, **[F-PROPA]** for iterative state reachability determination, and **[F-SUM]** for applying standard context-sensitive summaries [46] at callsites. This involves adding summary edges $n_1 \xrightarrow{\text{balanced}} n_2$ to encapsulate inter-procedural reachability, thereby streamlining reachability computations for method M .

$$\begin{array}{c}
 \frac{n_1 \xrightarrow{\hat{c}} \text{this}^M \in E}{\text{this}^M \in R(\text{flows}) \quad \text{flows} \in R^{-1}(\text{this}^M)} \quad [\text{F-INIT}] \\
 \frac{n_1 \xrightarrow{\ell} n_2 \in E \quad q_1 \in R^{-1}(n_1) \quad \delta(q_1, \ell) = q_2}{n_2 \in R(q_2) \quad q_2 \in R^{-1}(n_2)} \quad [\text{F-PROPA}] \\
 \frac{n_1 \xrightarrow{\hat{c}} \text{this}^M \in E \quad \text{this}^M \xrightarrow{\hat{c}} n_2 \in E \quad \overline{\text{flows}} \in R^{-1}(\text{this}^M)}{n_1 \xrightarrow{\text{balanced}} n_2 \in E} \quad [\text{F-SUM}]
 \end{array}$$

■ **Figure 11** Rules for conducting P3CTX over $G = (N, E)$.

► **Theorem 7.** *kCFA (performed in terms of the rules in Figure 1) produces exactly the same points-to information when performed with selective context-sensitivity under P3CTX.*

Proof. Follows from the facts that (1) Equation (21) provides necessary conditions for supporting selective context-sensitivity, (2) L_{DCR} provides a specification of kCFA with CFL-reachability for callgraph construction, (3) $L_{D^r C} \supseteq L_{DCR}$, and (4) [F-INIT] has weakened CS-C1 by starting from the `this` variable of every method instead of every object O . ◀

The worst-case time complexity of P3CTX in analyzing a program on $G = (N, E)$ is $O(|E| \times |Q|)$, which is linear to $|E|$ as $|Q|$ (the number of states in our DFA) is a constant.

4.2 Evaluation

We demonstrate that P3CTX significantly speeds up kCFA while maintaining precision. Compared to non-precision-preserving pre-analyses, SELECTX [33] and ZIPPER [29], P3CTX excels in achieving more efficient precision trade-offs in certain application scenarios.

4.2.1 Experimental Setup

We implemented *kCFA* (Figure 1) and P3CTX (Figure 11) in SOOT [59], using its context-insensitive pointer analysis, SPARK [26], for PAG construction. To compare P3CTX with SELECTX and ZIPPER, we used their existing implements from the SELECTX artifact [34]. Our evaluation follows pointer analysis standards [35, 33, 32, 42, 58, 14, 16], including using TAMIFLEX [4] for Java reflection, SOOT’s native code summaries, and context-insensitive analysis for special objects like strings and exceptions, distinguished per dynamic type.

We selected a set of 13 benchmarks from the DaCapo benchmark suite (latest version 6cf0380) [3] along with a large Java library (JRE1.8.0_31). We excluded `jython` because both *kCFA* and *P-kCFA* could not scale this benchmark due to its overly conservative reflection log [57]. Our artifact is publicly available at [19].

Our experiments were conducted on an Intel(R) Xeon(R) W-2245 3.90GHz machine with 512GB of RAM, operating under Ubuntu 20.04.3 LTS (Focal Fossa).

4.2.2 Results

Table 2 presents the results for *kCFA* and its three accelerated variants: *P-kCFA* (by P3CTX), *S-kCFA* (by SELECTX), and *Z-kCFA* (by ZIPPER), along with SPARK for comparison purposes, focusing on $k \in \{1, 2\}$. For $k \geq 3$, *kCFA* is unscalable for all 13 programs under a 12-hour budget and thus has never been considered in the literature [33, 42, 29, 30, 58, 50, 20, 57].

Table 2 Main analysis results. The analysis times for P - k CFA, S - k CFA, and Z - k CFA are given as $x(y)$, where x is the pointer analysis time and y is the pre-analysis time (in seconds). For all metrics, smaller is better.

Program	Metrics	SPARK	1CFA	P -1CFA	S -1CFA	Z -1CFA	2CFA	P -2CFA	S -2CFA	Z -2CFA
avrora	Time(secs)	6.6	18.0	4.7 (1.2)	3.1 (21.5)	2.8 (4)	577.1	142.5 (1.2)	16.8 (21.6)	11.2 (4)
	#Call Edges	57509	55267	55267	55267	55403	54505	54505	54506	54662
	#Fail Casts	1197	931	931	931	965	890	890	895	942
	#Alias Pairs	22327	13700	13700	13700	13703	13268	13268	13280	13547
	Avg PTS	36.19	25.87	25.87	25.87	26.48	24.78	24.78	24.80	25.47
batik	Time(secs)	30.9	81.0	28.0 (4.7)	25.3 (169.5)	23.1 (243)	1473.9	466.5 (4.8)	271.1 (174.4)	276.5 (234)
	#Call Edges	171409	151995	151995	151997	152025	147428	147428	147430	150549
	#Fail Casts	4573	3709	3709	3709	3713	3485	3485	3490	3620
	#Alias Pairs	68130	38005	38005	38005	38012	32288	32288	32300	33295
	Avg PTS	114.43	71.67	71.67	71.67	71.71	66.65	66.65	66.65	68.21
eclipse	Time(secs)	14.8	48.7	23.3 (2.0)	20.1 (54.6)	19.7 (14)	1221.1	331.0 (2.0)	171.8 (56.8)	143.9 (14)
	#Call Edges	110089	97960	97960	98000	98052	93662	93662	93703	93746
	#Fail Casts	2896	2470	2470	2471	2474	2322	2322	2328	2337
	#Alias Pairs	107389	58489	58489	58500	58504	51404	51404	51427	51716
	Avg PTS	101.12	63.49	63.49	63.47	63.80	59.28	59.28	59.26	59.64
fop	Time(secs)	76.0	318.8	123.1 (10.6)	113.1 (603.8)	104.0 (355)	6019.6	2399.6 (10.8)	1901.7 (604.5)	1405.1 (354)
	#Call Edges	358738	325547	325547	325551	325591	313954	313954	313958	321008
	#Fail Casts	9057	8226	8226	8228	8239	7931	7931	7938	8084
	#Alias Pairs	323628	277047	277047	277047	277065	267389	267389	267401	268943
	Avg PTS	233.48	141.19	141.19	141.19	141.25	132.98	132.98	132.98	135.43
h2	Time(secs)	16.1	75.7	18.5 (2.9)	15.8 (74.1)	14.3 (40)	6406.8	4164.6 (2.8)	3807.8 (74.4)	3127.4 (39)
	#Call Edges	144711	135775	135775	135782	135806	134234	134234	134241	134274
	#Fail Casts	2880	2477	2477	2477	2482	2398	2398	2404	2433
	#Alias Pairs	77978	39209	39209	39209	39236	33331	33331	33351	33632
	Avg PTS	72.61	34.61	34.61	34.61	34.68	32.63	32.63	32.64	33.20
luindex	Time(secs)	18.5	41.0	24.0 (1.9)	22.6 (48.1)	20.1 (8)	829.1	232.3 (1.9)	109.0 (48.2)	82.3 (8)
	#Call Edges	85850	79431	79431	79431	79602	78190	78190	78190	78404
	#Fail Casts	1726	1359	1359	1360	1376	1286	1286	1292	1314
	#Alias Pairs	50530	32905	32905	32905	32908	31795	31795	31807	32083
	Avg PTS	53.10	24.75	24.75	24.75	24.87	23.04	23.04	23.04	23.15
lusearch	Time(secs)	5.3	12.6	3.5 (1.0)	2.3 (13.9)	1.9 (3)	414.0	129.3 (1.0)	9.6 (13.9)	7.1 (3)
	#Call Edges	45285	43117	43117	43117	43198	42412	42412	42412	42516
	#Fail Casts	955	702	702	702	719	660	660	665	696
	#Alias Pairs	20382	11693	11693	11693	11696	11263	11263	11275	11542
	Avg PTS	31.38	20.73	20.73	20.74	20.85	19.73	19.73	19.75	19.94
pmd	Time(secs)	20.3	109.5	42.6 (3.0)	37.2 (139.1)	35.9 (25)	16006.8	13715.8 (3.0)	13671.4 (139.1)	9356.3 (25)
	#Call Edges	159395	153150	153150	153150	153387	152090	152090	152090	152242
	#Fail Casts	4702	4321	4321	4321	4325	4233	4233	4238	4263
	#Alias Pairs	114914	95977	95977	95977	95979	93083	93083	93095	93353
	Avg PTS	90.97	68.76	68.76	68.76	68.79	67.48	67.48	67.49	67.58
sunflow	Time(secs)	9.9	25.9	7.4 (1.8)	5.5 (46.4)	5.3 (9)	643.1	165.1 (1.7)	33.0 (45.9)	27.7 (9)
	#Call Edges	77346	74198	74198	74200	74241	73392	73392	73394	73685
	#Fail Casts	2192	1771	1771	1773	1776	1649	1649	1656	1684
	#Alias Pairs	36952	21670	21670	21670	21678	20703	20703	20715	21041
	Avg PTS	51.31	33.62	33.62	33.62	33.69	31.34	31.34	31.36	31.79
tomcat	Time(secs)	7.4	18.9	5.8 (1.3)	4.0 (20.8)	3.7 (4)	632.9	148.7 (1.3)	16.1 (20.8)	11.7 (4)
	#Call Edges	60649	57933	57933	57933	58024	57073	57073	57073	57369
	#Fail Casts	1264	959	959	960	963	874	874	880	910
	#Alias Pairs	30775	24504	24504	24504	24507	22202	22202	22214	22482
	Avg PTS	39.88	25.37	25.37	25.37	25.51	24.03	24.03	24.04	24.62
tradebeans	Time(secs)	8.7	25.9	7.6 (1.5)	5.6 (41.7)	5.2 (9)	737.4	166.5 (1.5)	30.2 (43.4)	18.2 (9)
	#Call Edges	70911	67742	67742	67742	67858	66814	66814	67018	67207
	#Fail Casts	1523	1132	1132	1132	1135	1054	1054	1059	1068
	#Alias Pairs	36256	27175	27175	27175	27178	25683	25683	25695	25950
	Avg PTS	47.67	31.80	31.80	31.80	31.87	29.95	29.95	29.98	30.18
tradesoap	Time(secs)	8.4	24.8	7.7 (1.6)	5.8 (46.8)	5.2 (9)	703.0	162.8 (1.5)	29.9 (49.4)	17.9 (9)
	#Call Edges	70911	67742	67742	67742	67858	66814	66814	67018	67207
	#Fail Casts	1523	1132	1132	1132	1135	1054	1054	1059	1068
	#Alias Pairs	36256	27175	27175	27175	27178	25683	25683	25695	25950
	Avg PTS	47.67	31.80	31.80	31.80	31.87	29.95	29.95	29.98	30.18
xalan	Time(secs)	8.5	27.3	7.4 (1.4)	5.5 (42.6)	5.0 (16)	702.8	162.3 (1.6)	34.2 (42.3)	26.0 (16)
	#Call Edges	69608	67132	67132	67132	67210	66360	66360	66360	66448
	#Fail Casts	1807	1473	1473	1473	1477	1419	1419	1424	1441
	#Alias Pairs	42119	28280	28280	28280	28283	27259	27259	27271	27539
	Avg PTS	45.29	29.41	29.41	29.41	29.47	28.29	28.29	28.30	28.41

4.2.2.1 Precision

Pointer analysis precision is gauged using four key metrics: (1) “#Call Edges”, indicating discovered call graph edges; (2) “#Fail Casts”, representing potential type cast failures; (3) “#Alias Pairs”, counting base variable pairs in stores and loads that may alias, excluding trivial must-aliases like direct assignments [10]; and (4) “Avg PTS”, the average number of objects pointed to by reachable local variables. Lower metric values signify higher precision.

For each metric M , M_{PTA} denotes the result obtained by PTA , where PTA denotes any pointer analysis in {SPARK, k CFA, P - k CFA, S - k CFA, Z - k CFA}. Let A - k CFA $\in \{P$ - k CFA, S - k CFA, Z - k CFA} be one of the three variants of k CFA such that A - k CFA is no less precise than SPARK but no more precise than k CFA. We define the precision loss of A - k CFA with respect to k CFA on metric M as:

$$\Delta_{A-k\text{CFA}}^M = \frac{(M_{\text{SPARK}} - M_{k\text{CFA}}) - (M_{\text{SPARK}} - M_{A-k\text{CFA}})}{M_{\text{SPARK}} - M_{k\text{CFA}}} = \frac{M_{A-k\text{CFA}} - M_{k\text{CFA}}}{M_{\text{SPARK}} - M_{k\text{CFA}}} \quad (29)$$

The precision gain from SPARK to k CFA is 100%. If A - k CFA matches k CFA in precision ($M_{A-k\text{CFA}} = M_{k\text{CFA}}$), then $\Delta_{A-k\text{CFA}}^M = 0\%$, indicating no precision loss in A - k CFA. Conversely, if A - k CFA reverts to SPARK’s precision ($M_{A-k\text{CFA}} = M_{\text{SPARK}}$), $\Delta_{A-k\text{CFA}}^M = 100\%$, reflecting a complete loss of k CFA’s precision advantage.

P - k CFA retains precision, matching k CFA across all metrics in 13 benchmarks, supported by Theorem 7 and Table 2. S - k CFA, leveraging L_{FC} for context-sensitivity, has small average precision losses of 0.8%, 1.2%, 0.1%, and 0.1% in “#Call Edges”, “#Fail Casts”, “#Alias Pairs”, and “Avg PTS”, respectively, at $k = 2$. However, for “#Call Edges”, S - 2 CFA incurs a 5% precision loss in both `tradebeans` and `tradesoap`. Conversely, Z - k CFA experiences higher average precision losses of 6.2%, 8.1%, 2.2%, and 2.0% for the same metrics at $k = 2$, attributed to ZIPPER’s use of pattern-based heuristics for context-sensitivity decisions.

To explore S - 2 CFA’s precision loss in `tradebeans` (Figure 12), it is noted that S - 2 CFA fails to identify the call in line 15 as monomorphic, unlike P - 2 CFA. When `put()` is invoked on a `TreeMap` object, a virtual call `compare()` occurs on the `comparator` object stored in the `TreeMap` object. With 2 CFA, `put()` is analyzed under contexts [L1] and [L2]. Under [L1], `cmp` links to `CMP1` and `k` to `I`, leading to `compare()` from line 10 to be invoked under [L3, L1]. Under [L2], `cmp` points to `CMP2` and `k` to `S1`, calling `compare()` from line 14 under [L3, L2], making `o1` point uniquely to `S1`. Thus, the virtual call in line 15 invokes only the `toString()` method defined in `java.lang.String`.

SELECTX, using L_{FC} , treats `cmp` and `k` in `put()` as context-insensitive, violating **CS-C3** in Equation (21). With S - 2 CFA, `o1` erroneously points to both `I` and `S1` under [L3, L2], leading to a polymorphic call in line 15. In contrast, P3CTX with L_{DCR} treats these as context-sensitive, adhering to **CS-C3**, resulting in `o1` pointing only to `S1` and ensuring a monomorphic call in line 15. This change prevents a 5% precision loss in “#Call Edges”, potentially enhancing critical software security analyses.

4.2.2.2 Efficiency

In Table 2, the efficiency of a pointer analysis is gauged by the time required in analyzing a program. This includes time for both the pointer analysis and the corresponding pre-analysis in each k CFA variant, denoted as A - k CFA ($A \in \{P, S, Z\}$). For $k = 1$ and $k = 2$, pre-analysis is done separately, causing slight differences in pre-analysis times for the same program. SPARK’s time is not included, as its results are shared by all three pre-analyses.

```

1 class TreeMap {
2     Comparator comparator;
3     TreeMap(Comparator cmp1) { this.comparator = cmp1; }
4     void put(Object k, Object v) {
5         Comparator cmp = this.comparator;
6         int i = cmp.compare(k, ...); // L3
7     }
8 // in java.lang.String
9 class CaseInsensitiveComparator implements Comparator {
10    int compare(String p1, String p2) { return 0; }
11 }
12 // in org.apache.geronimo.main
13 class StringComparator implements Comparator {
14    int compare(Object o1, Object o2) {
15        String s1 = o1.toString(); // #Call Edges?
16        return s1.compareTo(o2.toString());
17 }
18 void main() {
19     Comparator cmp1 = new CaseInsensitiveComparator(); // CMP1
20     Comparator cmp2 = new StringComparator(); // CMP2
21     TreeMap map1 = new TreeMap(cmp1); // M1
22     TreeMap map2 = new TreeMap(cmp2); // M2
23     Integer x = new Integer(1); // I
24     String y = new String(); // S1
25     z = new String(); // S2
26     map1.put(x, z); // L1
27     map2.put(y, z); // L2
28 }

```

Figure 12 An example abstracted from `tradebeans` and JDK8 to illustrate why SELECTX is not precision-preserving (by applying L_{FC} to determine precision-critical variables/objects in a program).

Table 2 reveals that P3CTX, SELECTX, and ZIPPER significantly boost k CFA for $k = 2$. Z-2CFA leads with $1.7\times$ to $41.0\times$ speedups, averaging $10.9\times$. S-2CFA ranges from $1.2\times$ to $17.6\times$, averaging $6.0\times$. P3CTX increases speeds from $1.2\times$ to $4.4\times$, averaging $3.2\times$. At $k = 1$, P3CTX performs best due to lower pre-analysis overhead and faster 1CFA. ZIPPER moderately improves 1CFA for most programs, but less effectively than P3CTX. SELECTX slows down 1CFA when including pre-analysis time. For P -1CFA, speedups range from $1.6\times$ to $3.5\times$, averaging $2.6\times$. Z-1CFA sees $0.3\times$ to $2.6\times$ speedups, averaging $1.5\times$. S-1CFA shows no gains, with $0.4\times$ to $0.8\times$ speedups, averaging $0.6\times$.

When assessing the precision and efficiency of P - k CFA, S - k CFA, and Z - k CFA, several key insights emerge. For tasks where precision is paramount, such as in software security analysis, P - k CFA emerges as the superior choice. It offers a speed advantage without compromising the precision inherent to k CFA. In contexts where the precision of 1CFA is needed, but with greater efficiency, P -1CFA is the standout option. It surpasses both S -1CFA and Z -1CFA in terms of speed while retaining the precision level of 1CFA. Finally, for applications requiring pointer analysis at the precision level of 2CFA, the recommendation depends on the user's priorities: Z -2CFA for those valuing efficiency above precision, S -2CFA for those who prioritize efficiency but can accept minor precision loss, and P -2CFA for those who deem precision crucial but also desire increased speed.

5 Related Work

In this section, we focus exclusively on prior work that is directly relevant to our study.

CFL-Reachability. CFL-reachability, introduced in program analysis for inter-procedural dataflow analysis [46, 44], has been applied in tackling various problems such as pointer analysis [54, 53, 64, 61, 62, 48, 63, 35, 32], information flow [37, 28, 36], and type inference [43, 41]. Traditionally, *kCFA*'s CFL-reachability formulation [53, 62, 48] relies on a separate call graph construction algorithm, either pre-applied or on-the-fly. This paper introduces L_{DCR} , a new CFL-reachability formulation for *kCFA*, integrating built-in call graph construction. An earlier attempt to address the same problem by Sridharan [52] is sound but less precise than L_{DCR} due to the lack of L_R . Without L_R , a context used for parameter passing at a virtual callsite can be incorrectly restored as a different context after finding the dispatched method and returning to the same callsite (as in Figure 9).

Another line of research on CFL-reachability focuses on its computational complexity. Generally, the all-pairs CFL-reachability problem can be resolved in $O(m^3n^3)$ time, where m is the CFL grammar size and n is the graph node count. Kodumal et al. [23] efficiently solved Dyck-CFL-reachability in $O(mn^3)$. Chaudhuri [7] later optimized the general CFL-reachability algorithm to subcubic time using the Four Russians' Trick [24]. Zhang et al. [63] demonstrated that bidirected Dyck-CFL reachability could be solved in $O(n + p \log p)$ (with p being the graph edge count), noting that reachability in a bidirected graph forms an equivalence relation. This complexity was further reduced to $O(p + n \cdot \alpha(n))$ in [6], where $\alpha(n)$ is the inverse Ackermann function. This paper introduces P3CTX, an L_{DCR} -enabled pre-analysis for accelerating *kCFA*, linear in terms of the number of PAG edges in the program's PAG and preserving precision.

A CFL-reachability-based formulation recently proposed for object-sensitive pointer analysis [35, 38, 39] naturally includes call graph construction, as it uses receiver objects as context elements. However, integrating call graph construction into callsite-sensitive analyses using the traditional CFL-reachability framework [53, 62, 48] is challenging, as detailed in Section 2. An earlier attempt [52] was sound but lacked precision, particularly in restoring contexts correctly after method dispatch and return at virtual callsites, as shown in Figure 9. L_{DCR} is the first known solution to effectively integrate call graph construction into CFL-reachability for callsite-sensitive analyses.

Selective Context-sensitivity. In the realm of pointer analysis acceleration, three primary approaches exist: pattern-based [51, 12, 29, 30], data-driven [21, 20], and CFL-reachability-guided [35, 33, 14, 13]. By exploiting CFL-reachability, EAGLE [35, 32], TURNER [14], CONCH [16, 18], and DEBLOATERX [13] represent recent efforts in accelerating object-sensitive pointer analysis [39]. SELECTX [33] marks the initial CFL-reachability-based effort to accelerate *kCFA*, but it lacks precision preservation due to its reliance on L_{FC} [53]. This paper introduces P3CTX, the first precision-preserving pre-analysis for *kCFA*, grounded in L_{DCR} .

6 Conclusion

We have introduced L_{DCR} , a new CFL-reachability formulation for supporting *k*-callsite-based context-sensitive pointer analysis (*kCFA*), featuring a unique built-in call graph construction to effectively handle dynamic dispatch. To demonstrate its utility, we have also introduced P3CTX, which is developed based on L_{DCR} , to enhance the performance of *kCFA* while preserving its precision. We hope that L_{DCR} can provide some new insights on understanding *kCFA* and its demand-driven forms [54, 53, 62], potentially inspiring novel algorithmic advancements. Future explorations include applying L_{DCR} to selective context sensitivity and extending its application to areas such as library-code summarization [48, 56, 8] and information flow analysis [28, 36].

References

- 1 Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Copenhagen, 1994.
- 2 David F Bacon and Peter F Sweeney. Fast static analysis of c++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 324–341, New York, NY, USA, 1996. Association for Computing Machinery.
- 3 Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, New York, NY, USA, 2006. Association for Computing Machinery.
- 4 Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 241–250, Honolulu, HI, USA, 2011. IEEE.
- 5 Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 243–262, New York, NY, USA, 2009. Association for Computing Machinery.
- 6 Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. Optimal Dyck reachability for data-dependence and alias analysis. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–30, 2017.
- 7 Swarat Chaudhuri. Subcubic algorithms for recursive state machines. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 159–169, New York, NY, USA, 2008. Association for Computing Machinery.
- 8 Yifan Chen, Chenyang Yang, Xin Zhang, Yingfei Xiong, Hao Tang, Xiaoyin Wang, and Lu Zhang. Accelerating program analyses in datalog by merging library facts. In *International Static Analysis Symposium*, pages 77–101, Cham, 2021. Springer, Springer International Publishing.
- 9 Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*, pages 77–101, Berlin, Heidelberg, 1995. Springer, Springer Berlin Heidelberg.
- 10 Yu Feng, Xinyu Wang, Isil Dillig, and Thomas Dillig. Bottom-up context-sensitive pointer analysis for Java. In *Programming Languages and Systems: 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30–December 2, 2015, Proceedings*, pages 465–484, Cham, 2015. Springer International Publishing.
- 11 David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(6):685–746, 2001.
- 12 Behnaz Hassanshahi, Raghavendra Kagalavadi Ramesh, Padmanabhan Krishnan, Bernhard Scholz, and Yi Lu. An efficient tunable selective points-to analysis for large codebases. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 13–18, New York, NY, USA, 2017. Association for Computing Machinery.
- 13 Dongjie He, Yujiang Gui, Wei Li, Yonggang Tao, Changwei Zou, Yulei Sui, and Jingling Xue. A container-usage-pattern-based context debloating approach for object-sensitive pointer analysis. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA2):971–1000, 2023.
- 14 Dongjie He, Jingbo Lu, Yaoqing Gao, and Jingling Xue. Accelerating object-sensitive pointer analysis by exploiting object containment and reachability. In *Proceedings of the 35th European Conference on Object-Oriented Programming (ECOOP 2021)*, pages 18:1–18:31, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

- 15 Dongjie He, Jingbo Lu, and Jingling Xue. A CFL-Reachability Formulation of Callsite-Sensitive Pointer Analysis with Built-in On-the-Fly Call Graph Construction (Artifact). Software, version 1.0. (visited on 2024-08-27). URL: <https://doi.org/10.5281/zenodo.11061892>.
- 16 Dongjie He, Jingbo Lu, and Jingling Xue. Context debloating for object-sensitive pointer analysis. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 79–91, New York, NY, USA, 2021. IEEE. doi:10.1109/ASE51524.2021.9678880.
- 17 Dongjie He, Jingbo Lu, and Jingling Xue. Qilin: A new framework for supporting fine-grained context-sensitivity in Java pointer analysis. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:29, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2022.30.
- 18 Dongjie He, Jingbo Lu, and Jingling Xue. IFDS-based context debloating for object-sensitive pointer analysis. *ACM Transactions on Software Engineering and Methodology*, 2023.
- 19 Dongjie He, Jingbo Lu, and Jingling Xue. A CFL-reachability formulation of callsite- sensitive pointer analysis with built-in on-the- fly call graph construction (artifact), July 2024. doi: 10.5281/zenodo.11061892.
- 20 Minseok Jeon, Sehun Jeong, and Hakjoo Oh. Precise and scalable points-to analysis via data- driven context tunneling. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–29, 2018.
- 21 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.
- 22 George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 423–434, New York, NY, USA, 2013. Association for Computing Machinery.
- 23 John Kodumal and Alex Aiken. The set constraint/CFL reachability connection in practice. *ACM Sigplan Notices*, 39(6):207–218, 2004.
- 24 VL Arlazarov EA Dinic MA Kronrod and IA Faradzev. On economic construction of the transitive closure of a directed graph. In *Dokl. Acad. Nauk SSSR*, pages 487–88, 1970.
- 25 Michael John Latta. *The intersection of context-free languages*. PhD thesis, University of Texas at Austin, USA, 1993. URL: <https://www.proquest.com/docview/304086568?pq-origsite=gscholar&fromopenview=true>.
- 26 Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction*, pages 153–169, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.
- 27 Ondřej Lhoták and Laurie Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 18(1):1–53, 2008.
- 28 Yuanbo Li, Qirun Zhang, and Thomas Reps. Fast graph simplification for interleaved Dyck- reachability. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 780–793, New York, NY, USA, 2020. Association for Computing Machinery.
- 29 Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. Precision-guided context sensitivity for pointer analysis. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–29, 2018.
- 30 Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. A principled approach to selective context sensitivity for pointer analysis. *ACM Transactions on Programming Languages and Systems*, 42(TOPLAS):1–40, 2020.
- 31 Leonard Y Liu and Peter Weiner. An infinite hierarchy of intersections of context-free languages. *Mathematical systems theory*, 7:185–192, 1973. doi:10.1007/BF01762237.

- 32 Jingbo Lu, Dongjie He, and Jingling Xue. Eagle: CFL-reachability-based precision-preserving acceleration of object-sensitive pointer analysis with partial context sensitivity. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(4):1–46, 2021.
- 33 Jingbo Lu, Dongjie He, and Jingling Xue. Selective context-sensitivity for k-CFA with CFL-reachability. In *International Static Analysis Symposium*, pages 261–285, Cham, 2021. Springer, Springer International Publishing.
- 34 Jingbo Lu, Dongjie He, and Jingling Xue. Selective context-sensitivity for k-CFA with CFL-reachability (artifact), July 2021. doi:10.5281/zenodo.4732680.
- 35 Jingbo Lu and Jingling Xue. Precision-preserving yet fast object-sensitive pointer analysis with partial context sensitivity. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.
- 36 Ana Milanova. FlowCFL: generalized type-based reachability analysis: graph reduction and equivalence of CFL-based and type-based reachability. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020.
- 37 Ana Milanova, Wei Huang, and Yao Dong. CFL-reachability and context-sensitive integrity types. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*, pages 99–109, New York, NY, USA, 2014. Association for Computing Machinery.
- 38 Ana Milanova, Atanas Rountev, and Barbara G Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 1–11, New York, NY, USA, 2002. Association for Computing Machinery.
- 39 Ana Milanova, Atanas Rountev, and Barbara G Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, 2005.
- 40 Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319, New York, NY, USA, 2006. Association for Computing Machinery.
- 41 Polyvios Pratikakis, Jeffrey S Foster, and Michael Hicks. Existential label flow inference via CFL reachability. In *International Static Analysis Symposium*, pages 88–106, Berlin, Heidelberg, 2006. Springer, Springer Berlin Heidelberg.
- 42 Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. User-guided program reasoning using bayesian inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 722–735, New York, NY, USA, 2018. Association for Computing Machinery.
- 43 Jakob Rehof and Manuel Fähndrich. Type-based flow analysis: from polymorphic subtyping to CFL-reachability. *ACM SIGPLAN Notices*, 36(3):54–66, 2001.
- 44 Thomas Reps. Program analysis via graph reachability. *Information and software technology*, 40(11-12):701–726, 1998.
- 45 Thomas Reps. Undecidability of context-sensitive data-dependence analysis. *ACM Transactions on Programming Languages and Systems*, 22(1):162–186, 2000.
- 46 Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, New York, NY, USA, 1995. Association for Computing Machinery.
- 47 Barbara G Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *International Conference on Compiler Construction*, pages 126–137, Berlin, Heidelberg, 2003. Springer, Springer Berlin Heidelberg.
- 48 Lei Shang, Xinwei Xie, and Jingling Xue. On-demand dynamic summary-based points-to analysis. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 264–274, New York, NY, USA, 2012. Association for Computing Machinery.
- 49 Olin Grigsby Shivers. *Control-flow analysis of higher-order languages or taming lambda*. PhD thesis, Carnegie Mellon University, 1991. CMU-CS-91-145.

- 50 Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 17–30, New York, NY, USA, 2011. Association for Computing Machinery.
- 51 Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: context-sensitivity, across the board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 485–495, New York, NY, USA, 2014. Association for Computing Machinery.
- 52 Manu Sridharan. *Refinement-based program analysis tools*. University of California, Berkeley, 2007.
- 53 Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 387–400, New York, NY, USA, 2006. Association for Computing Machinery.
- 54 Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 59–76, New York, NY, USA, 2005. Association for Computing Machinery.
- 55 Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. *ACM SIGPLAN Notices*, 35(10):264–280, 2000.
- 56 Hao Tang, Xiaoyin Wang, Lingming Zhang, Bing Xie, Lu Zhang, and Hong Mei. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 83–95, New York, NY, USA, 2015. Association for Computing Machinery.
- 57 Rei Thiessen and Ondřej Lhoták. Context transformations for pointer analysis. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 263–277, New York, NY, USA, 2017. Association for Computing Machinery.
- 58 Tian Tan, Yue Li and Jingling Xue. Efficient and precise points-to analysis: modeling the heap by merging equivalent automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 278–291, New York, NY, USA, 2017. Association for Computing Machinery.
- 59 Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., USA, 2010.
- 60 WALA. WALA: T.J. Watson Libraries for Analysis, 2024. URL: <https://github.com/wala/WALA>.
- 61 Guoqing Xu, Atanas Rountev, and Manu Sridharan. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *European Conference on Object-Oriented Programming*, pages 98–122, Berlin, Heidelberg, 2009. Springer, Springer Berlin Heidelberg.
- 62 Dacong Yan, Guoqing Xu, and Atanas Rountev. Demand-driven context-sensitive alias analysis for Java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 155–165, New York, NY, USA, 2011. Association for Computing Machinery.
- 63 Qirun Zhang, Michael R Lyu, Hao Yuan, and Zhendong Su. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 435–446, New York, NY, USA, 2013. Association for Computing Machinery.
- 64 Xin Zheng and Radu Rugina. Demand-driven alias analysis for c. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 197–208, New York, NY, USA, 2008. Association for Computing Machinery.

Fearless Asynchronous Communications with Timed Multiparty Session Protocols

Ping Hou 

University of Oxford, UK

Nicolas Lagaillardie 

Imperial College London, UK

Nobuko Yoshida 

University of Oxford, UK

Abstract

Session types using *affinity* and *exception handling* mechanisms have been developed to ensure the communication safety of protocols implemented in concurrent and distributed programming languages. Nevertheless, current affine session types are inadequate for specifying real-world asynchronous protocols, as they are usually imposed by *time constraints* which enable *timeout exceptions* to prevent indefinite blocking while awaiting valid messages. This paper proposes the first formal integration of *affinity*, *time constraints*, *timeouts*, and *time-failure handling* based on multiparty session types for supporting reliability in asynchronous distributed systems. With this theory, we statically guarantee that asynchronous timed communication is deadlock-free, communication safe, while being *fearless* – never hindered by timeout errors or abrupt terminations.

To implement our theory, we introduce *MultiCrusty^T*, a RUST toolchain designed to facilitate the implementation of safe affine timed protocols. *MultiCrusty^T* leverages generic types and the `time` library to handle timed communications, integrated with optional types for affinity. We evaluate *MultiCrusty^T* by extending diverse examples from the literature to incorporate time and timeouts. We also showcase the *correctness by construction* of our approach by implementing various real-world use cases, including protocols from the Internet of Remote Things domain and real-time systems.

2012 ACM Subject Classification Software and its engineering → Software usability; Software and its engineering → Concurrent programming languages; Theory of computation → Process calculi

Keywords and phrases Session Types, Concurrency, Time Failure Handling, Affinity, Timeout, Rust

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.19

Related Version Full Version: <https://arxiv.org/abs/2406.19541> [19]

Supplementary Material Software (Source Code): https://github.com/NicolasLagaillardie/mpst_rust_github, archived at [swsh:1:dir:08181be2bf9b8bd74ec08356de274ee93a9c7db9](https://doi.org/10.4230/zenodo.1021811)

Software (ECOOP 2024 Artifact Evaluation approved artifact):

<https://doi.org/10.4230/DARTS.10.2.10>

Funding Work supported by: EPSRC EP/T006544/2, EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/2, EP/N028201/1, EP/T014709/2, EP/V000462/1, EP/X015955/1, NCSS/EPSRC VeTSS, and Horizon EU TaRDIS 101093006.

Acknowledgements We thank the anonymous reviewers for their useful comments and suggestions.

1 Introduction

Background. The growing prevalence of distributed programming has emphasised the significance of prioritising *reliability* in distributed systems. Dedicated research efforts focus on enhancing reliability through the study and modelling of failures. This research enables the design of more resilient distributed systems, capable of effectively handling failures and ensuring reliable operation.

 © Ping Hou, Nicolas Lagaillardie, and Nobuko Yoshida;
licensed under Creative Commons License CC-BY 4.0
38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 19; pp. 19:1–19:30



A lightweight, type-based methodology, which ensures basic reliability – *safety* in distributed communication systems, is *session types* [16]. This type discipline is further advanced by *Multiparty Session Types* (MPST) [17, 18], which enable the specification and verification of communication protocols among multiple message-passing processes in concurrent and distributed systems. MPST ensure that protocols are designed to prevent common safety errors, i.e. deadlocks and communication mismatches during interactions among many participants [17, 18, 37]. By adhering to a specified MPST protocol, participants (a.k.a. end-point programs) can communicate reliably and efficiently. From a practical perspective, MPST have been implemented in various programming languages [5, 26, 28, 32, 40, 42], facilitating their applications and providing safety guarantees in real-world programs.

Nevertheless, tackling the challenges of unreliability and failures remains a significant issue for session types. Most session type systems operate under the assumption of flawless and reliable communication without failures. To address this limitation, recent works [31, 14, 15, 28] have developed *affine session types* by incorporating the *affinity* mechanism that explicitly accounts for and handles unreliability and failures within session type systems. Unlike linear types that must be used *exactly* once, affine types can be used *at most* once, enabling the safe dropping of subsequent types and the premature termination of a session in the presence of protocol execution errors.

In most real-life devices and platforms, communications are predominantly asynchronous: inner tasks and message transfers may take time. When dealing with such communications, it becomes crucial to incorporate *time constraints* and implement *timeout failure handling* for each operation. This is necessary to avoid potential blockages where a process might wait indefinitely for a message from a non-failed process. While various works, as explained later, address time conditions and timeouts in session types, it is surprising that none of the mentioned works on affine session types tackles timeout failures during protocol execution.

This Paper. We introduce a new framework, *affine timed multiparty session types* (ATMP), to address the challenges of timeouts, disconnections and other failures in asynchronous communications:

- (1) We propose ATMP, an extension of asynchronous MPST that incorporates time specifications, affinity, and mechanisms for handling exceptions, thus facilitating effective management of failures, with a particular focus on timeouts. Additionally, we demonstrate that properties from MPST, i.e. *type safety*, *protocol conformance*, and *deadlock-freedom*, are guaranteed for well-typed processes, even in the presence of timeouts and their corresponding handling mechanism;
- (2) We present *MultiCrusty^T*, our RUST toolchain designed for building asynchronous timed multiparty protocols under ATMP: *MultiCrusty^T* enables the implementation of protocols adhering to the properties of ATMP.

The primary focus of ATMP lies in effectively *handling* timeouts during process execution, in contrast to the approaches in [4, 3], which aim to completely *avoid* time failures. Bocchi et al. [4] introduce time conditions in MPST to ensure precise timing in communication protocols, while their subsequent work [3] extends *binary* timed session types to incorporate timeouts, allowing for more robust handling of time constraints. Yet, they adopt strict requirements to *prevent* timeouts. In [4], *feasibility* and *wait-freedom* are required in their protocol design. Feasibility requires precise time specifications for protocol termination, while wait-freedom prohibits overlapping time windows for senders and receivers in a protocol, which is not practical in real-world applications. Similarly, in [3], strong conditions including *progress* of an entire set of processes and *urgent receive* are imposed. The progress property is usually *undecidable*, and the urgent receive condition, which demands immediate message reception upon availability, is infeasible with asynchronous communication.

Recently, [30] proposes the inclusion of timeout as the unique failure notation in MPST, offering flexibility in handling failures. Time also plays a role in *synchronous* communication systems, where [22] develops *rate-based binary* session types, ensuring *synchronous* message exchanges at the same *rate*, i.e. within the same time window. However, in both [30] and [22], time constraints are not integrated into types and static type checking, resulting in the specifications lacking the ability to guide time behaviour. Additionally, the model used in [22] assumes that all communications and computations are non-time-consuming, i.e. with zero time cost, making it unfeasible in distributed systems.

By the efficient integration of time and failure handling mechanisms in our framework, none of those impractical requirements outlined in [4, 3] is necessary. In ATMP, when a process encounters a timeout error, a mechanism for handling time failures is triggered, notifying all participants about the timeout, leading to the termination of those participants and ultimately ending the session. Such an approach guarantees that participants consistently reach the *end of the protocol*, as the communication session is entirely dropped upon encountering a timeout error. As a result, every process can terminate successfully, reducing the risk of indefinite blockages, even with timeouts. Additionally, in our system, time constraints over local clocks are incorporated with types to effectively model asynchronous timed communication, addressing the limitations in [30, 22].

Except for [22], the aforementioned works on timed session types focus more on theory, lacking implementations. To bridge this gap on the practical side, we provide *MultiCrusty^T*, a RUST implementation of ATMP designed for secure timed communications. *MultiCrusty^T* makes use of affine timed meshed channels, a communication data structure that integrates time constraints and clock utilisation. Our toolchain relies on macros and native generic types to ensure that asynchronous protocols are inherently *correct by construction*. In particular, *MultiCrusty^T* performs compile-time verification to guarantee that, at any given point in the protocol, each isolated pair of participants comprises one sender and one receiver with corresponding time constraints. Additionally, we employ affine asynchronous primitives and native optional types to effectively handle *runtime* timeouts and errors.

To showcase the capabilities and expressiveness of our toolchain, we evaluate *MultiCrusty^T* through examples from the literature, and further case studies including a remote data protocol from an Internet of Remote Things (IoRT) network [7], a servo web protocol from a web engine [38], and protocols from real-time systems such as Android motion sensor [2], PineTime smartwatch [35], and keyless entry [41]. Our comparative analysis with a RUST implementation of affine MPST without time [28] reveals that *MultiCrusty^T* exhibits minimal overhead while providing significantly strengthened property checks.

Structure. § 2 offers a comprehensive overview of our theory and toolchain. § 3 provides a session π -calculus for ATMP that incorporates timeout, affinity, asynchrony, and failure handling mechanisms. § 4 introduces an extended theory of asynchronous multiparty session types with time annotations. Additionally, we present a typing system for ATMP session π -calculus, and demonstrate the properties of typed processes. § 5 delves into the design and usage of *MultiCrusty^T*, our RUST implementation of ATMP. § 6 showcases the compilation and execution benchmarks of *MultiCrusty^T*, based on selected case studies. § 7 concludes the paper by discussing related work, and offering conclusions and potential future work. Full proofs, auxiliary material, and more details of *MultiCrusty^T* can be found in the full version of the paper [19]. Our toolchain and evaluation examples are available in an [artifact](#).

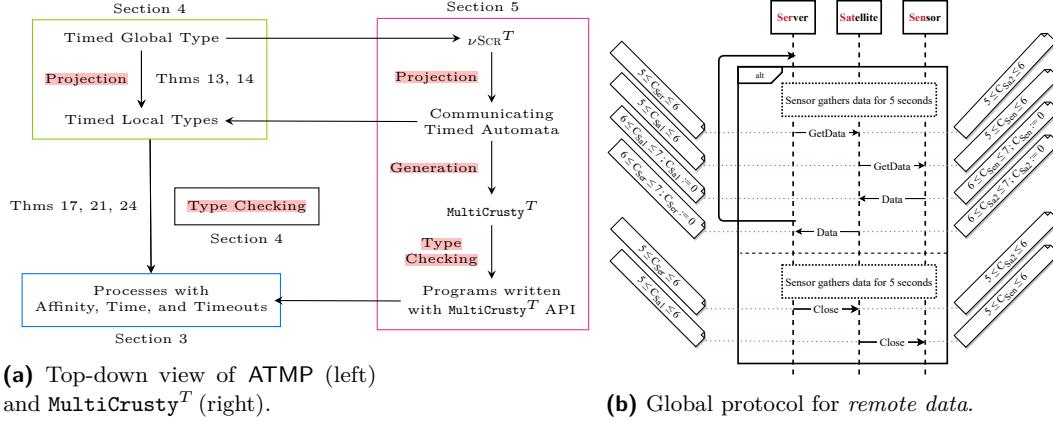


Figure 1 Overview of affine asynchronous communication with time.

2 Overview

In this section, we give an overview of affine timed multiparty session types (ATMP) and MultiCrusty^T, our toolchain for implementing affine timed asynchronous protocols. First, we share a real-world example inspiring our work on affine asynchronous timed communication.

Fig. 1b depicts our running example, *remote data*. This real-world scenario is sourced from a satellite-enabled Internet of Remote Things network [7], and describes data transmissions among a *Sensor* (*Sen*), a *Server* (*Ser*), and a *Satellite* (*Sat*): *Ser* aims to periodically retrieve data gathered by *Sen* via *Sat*. The protocol revolves around a loop initiated by *Ser*, which faces a decision: either retrieve data or end the protocol. In the former scenario, *Ser* requests data retrieval from *Sen* with a message labelled *GetData* via *Sat* within the time window of 5 and 6 time units, as indicated by clock constraints (i.e. $5 \leq C_{\text{Ser}} \leq 6$, where C_{Ser} is the clock associated with *Ser*). Upon receiving this request, *Sen* responds by sending the data with a message labelled *Data* to *Ser* through *Sat* within 6 and 7 time units, followed by clock resets denoted as reset predicates (i.e. $C_{\text{Ser}} := 0$, resetting the clock to 0). In the alternative branch, *Ser* sends a *Close* message to *Sat*, which is then forwarded to *Sen*, between 5 and 6 time units.

Our remote data protocol includes internal tasks that consume time, notably *Sen* requiring 5 time units to gather data before transmitting. In cases where our protocol lacks a specified timing strategy (i.e. no time requirements), and *Sen* cannot accomplish the data-gathering tasks, it results in indefinite blocking for *Sat* and *Ser* as they await the data. This could lead to undesirable outcomes, including partially processed data, data corruption, or incomplete transmission of processed data. Therefore, incorporating time constraints into communication protocols is imperative, as it better reflects real-world scenarios and ensures practical viability.

2.1 ATMP: Theory Overview

Our ATMP theory follows the *top-down* methodology [17, 18], enhancing asynchronous MPST with time features to facilitate timed global and local types. As shown in Fig. 1a (left), we specify multiparty protocols with time as *timed global types*. These timed global types are projected into *timed local types*, which are then used for type-checking processes with affine types, time, timeouts, and failure handling, written in a session calculus. As an example, we consider a simple communication scenario derived from remote data: the Satellite (*Sat*) communicates with the server (*Ser*) by sending a *Data* message (*Data*). Specifically, *Sat* needs to send the message between 6 and 7 time units and reset its clock afterwards, while *Ser* is expected to receive the message within the same time window and reset its clock accordingly.

Timed Types and Processes. This communication behaviour can be represented by the timed global type G :

$$\text{Sat} \rightarrow \text{Ser}: \{\text{Data}\{6 \leq C_{\text{Sat}} \leq 7, C_{\text{Sat}} := 0, 6 \leq C_{\text{Ser}} \leq 7, C_{\text{Ser}} := 0\}\}.end\}$$

where C_{Sat} and C_{Ser} denote the clocks of **Sat** and **Ser**, respectively. A global type represents a protocol specification involving multiple roles from a global standpoint.

Adhering to the MPST top-down approach, a timed global type is then *projected* onto timed local types, which describe communications from the perspective of individual roles. In our example, G is projected onto two timed local types, one for each role **Sat** and **Ser**:

$$T_{\text{Sat}} = \text{Ser} \oplus \text{Data}\{6 \leq C_{\text{Sat}} \leq 7, C_{\text{Sat}} := 0\}.end \quad T_{\text{Ser}} = \text{Sat} \& \text{Data}\{6 \leq C_{\text{Ser}} \leq 7, C_{\text{Ser}} := 0\}.end$$

Here T_{Sat} indicates that **Sat** sends (\oplus) the message **Data** to **Ser** between 6 and 7 time units and then immediately resets its clock C_{Sat} . Dually, T_{Ser} denotes **Ser** receiving ($\&$) the message from **Sat** within the same time frame and resetting its clock C_{Ser} .

In the final step of the top-down approach, we employ timed local types to conduct type-checking for processes, denoted as P_i , in the ATMP session calculus. Our session calculus extends the framework for *affine multiparty session types* (AMPST) [28] by incorporating processes that model time, timeouts, and asynchrony. In our example, T_{Sat} and T_{Ser} are used for the type-checking of $s[\text{Sat}]$ and $s[\text{Ser}]$, which respectively represent the channels (a.k.a. session endpoints) played by roles **Sat** and **Ser** in a multiparty session s , within the processes:

$$P_{\text{Sat}} = \text{delay}(C_1 = 6.5).s[\text{Sat}]^{0.4}[\text{Ser}] \oplus \text{Data.0} \quad P_{\text{Ser}} = \text{delay}(C_2 = 6).s[\text{Ser}]^{0.3}[\text{Sat}]\text{Data.0}$$

The Satellite process P_{Sat} waits for exactly 6.5 time units ($\text{delay}(C_1 = 6.5)$), then sends the message **Data** with a timeout of 0.4 time units ($s[\text{Sat}]^{0.4}[\text{Ser}] \oplus \text{Data}$), and becomes inactive (**0**). Meanwhile, the Server process P_{Ser} waits for 6 time units ($\text{delay}(C_2 = 6)$), then receives the message with a timeout of 0.3 time units ($s[\text{Ser}]^{0.3}[\text{Sat}]\text{Data}$), subsequently becoming inactive.

Solution to Stuck Processes Due to Time Failures. It appears that the parallel execution of P_{Sat} and P_{Ser} , $P_{\text{Sat}} \mid P_{\text{Ser}}$, cannot proceed further due to the disparity in timing requirements. Specifically, using the same session s , **Sat** sends the message **Data** to **Ser** between 6.5 and 6.9 time units, while **Ser** must receive it from **Sat** between 6 and 6.3 time units. This results in a stuck situation, as **Ser** cannot meet the required timing condition to receive the message.

Fortunately, in our system, timeout failures are allowed, which can be addressed by leveraging affine session types and their associated failure handling mechanisms. Back to our example, when $s[\text{Ser}]$ waits for 6 time units and cannot receive **Data** within 0.3 time units, a timeout failure is raised ($\text{timeout}[s[\text{Ser}]^{0.3}[\text{Sat}]\text{Data.0}]$). Furthermore, we apply our time-failure handling approach to manage this timeout failure, initiating the termination of the channel $s[\text{Ser}]$ and triggering the cancellation process of the session s ($s\#$). As a result, the process will successfully terminate by canceling (or killing) all usages of s within it.

Conversely, the system introduced in [4] enforces strict requirements, including *feasibility* and *wait-freedom*, on timed global types to prevent time-related failures in well-typed processes, thus preventing them from becoming blocked due to unsolvable timing constraints. Feasibility ensures the successful termination of each allowed partial execution, while wait-freedom guarantees that receivers do not have to wait if senders follow their time constraints. In our example, we start with a timed global type that is neither feasible nor wait-free, showcasing how our system effectively handles time failures and ensures successful process termination without imposing additional conditions on timed global types. In essence, reliance on feasibility and wait-freedom becomes unnecessary in our system, thanks to the inclusion of affinity and time-failure handling mechanisms.

```

1 struct Send<T>,
2 const CLOCK: char,
3 const START: i128,
4 const INCLUDE_START: bool,
5 const END: i128,
6 const INCLUDE_END: bool,
7 const RESET: char,
8 S>
1 struct Recv<T>,
2 const CLOCK: char,
3 const START: i128,
4 const INCLUDE_START: bool,
5 const END: i128,
6 const INCLUDE_END: bool,
7 const RESET: char,
8 S>
1 MeshedChannels<
2 Recv<Data,
3 'a',6,true,7,true,'a',End>,
4 Send<Data,
5 'b',6,true,7,true,'b',End>,
6 RoleSer<RoleSer<End>>,
7 NameSat,
8 >

```

(a) `Send` type. (b) `Recv` type. (c) `MeshedChannels` type for `Sat`.

Figure 2 Main types of MultiCrusty^T.

2.2 MultiCrusty^T: Toolchain Overview

To augment the theory, we introduce the MultiCrusty^T library, a toolchain for implementing communication protocols in RUST. MultiCrusty^T specifies protocols where communication operations must adhere to specific time limits (*timed*), allowing for *asynchronous* message reception and runtime handling of certain failures (*affine*). This library relies on two fundamental types: `Send` and `Recv`, representing message sending and receiving, respectively. Additionally, it incorporates the `End` type, signifying termination to close the connection. Figs. 2a and 2b illustrate the `Send` and `Recv` types respectively, used for sending and receiving messages of any thread-safe type (represented as `T` in Line 1). After sending or receiving a message, the next operation or continuation (`S` in Line 8) is determined, which may entail sending another message, receiving another message, or terminating the connection.

Similar to ATMP, each communication operation in MultiCrusty^T is constrained by specific time boundaries to avoid infinite waiting. These time bounds are represented by the parameters in Lines 2–7 of Fig. 2a, addressing scenarios where a role may be required to send after a certain time unit or receive between two specific time units. Consider the final communication operation in the first branch of Fig. 1b from `Sat`'s perspective. To remain consistent with § 2.1, the communication is terminated here instead of looping back to the beginning of the protocol. In this operation, `Sat` sends a message labelled `Data` to `Ser` between time units 6 and 7, with respect to its inner clock '`b`', and then terminates after resetting its clock. This can be implemented as: `Send<Data, 'b', 6, true, 7, true, 'b', End>`.

To enable multiparty communication in MultiCrusty^T, we use the `MeshedChannels` type, inspired by [28]. This choice is necessary as `Send` and `Recv` types are primarily designed for *binary* (peer-to-peer) communication. Within `MeshedChannels`, each binary channel pairs the owner role with another, establishing a mesh of communication channels that encompasses all participants. Fig. 2c demonstrates an example of using `MeshedChannels` for `Sat` in our running example: `Sat` receives a `Data` message from `Sen` (Line 2) and forwards it to `Ser` (Line 4) before ending all communications, following the order specified by the stack in Line 6.

Creating these types manually in RUST can be challenging and error-prone, especially because they represent the local perspective of each role in the protocol. Therefore, as depicted in Fig. 1a (right), MultiCrusty^T employs a top-down methodology similar to ATMP to generate local viewpoints from a global protocol, while ensuring the *correctness* of the generated types *by construction*. To achieve this, we extend the syntax of ν SCR [42], a language for describing multiparty communication protocols, to include time constraints, resulting in ν SCR^T. A timed global protocol represented in ν SCR^T is then projected onto local types, which are used for generating RUST types in MultiCrusty^T.

3 Affine Timed Multiparty Session Calculus

In this section, we formalise an affine timed multiparty session π -calculus, where processes are capable of performing time actions, raising timeouts, and handling failures. We start with the formal definitions of time constraints used in the paper.

Clock Constraint, Valuation, and Reset. Our time model is based on the timed automata formalism [1, 27]. Let \mathbf{C} denote a finite set of *clocks*, ranging over C, C', C_1, \dots , that take non-negative real values in $\mathbb{R}_{\geq 0}$. Additionally, let t, t', t_1, \dots be *time constants* ranging over $\mathbb{R}_{\geq 0}$. A *clock constraint* δ over \mathbf{C} is defined as:

$$\delta ::= \text{true} \mid C > \mathfrak{b} \mid C = \mathfrak{b} \mid \neg\delta \mid \delta_1 \wedge \delta_2$$

where $C \in \mathbf{C}$ and \mathfrak{b} is a *constant time bound* ranging over non-negative rationals $\mathbb{Q}_{\geq 0}$. We define $\text{false}, <, \geq, \leq$ in the standard way. For simplicity and consistency with our implementation (§ 5), we assume each clock constraint contains a *single* clock. Extending a clock constraint with *multiple* clocks is straightforward.

A *clock valuation* $\mathbb{V} : \mathbf{C} \rightarrow \mathbb{R}_{\geq 0}$ assigns time to each clock in \mathbf{C} . We define $\mathbb{V} + t$ as the valuation that assigns to each $C \in \mathbf{C}$ the value $\mathbb{V}(C) + t$. The *initial* valuation that maps all clocks to 0 is denoted as \mathbb{V}^0 , and the valuation that assigns a value of t to all clocks is denoted as \mathbb{V}^t . $\mathbb{V} \models \delta$ indicates that the constraint δ is satisfied by the valuation \mathbb{V} . Additionally, we use $\sqcup_{i \in I} \mathbb{V}_i$ to represent the overriding union of the valuations \mathbb{V}_i for $i \in I$.

A *reset predicate* λ over \mathbf{C} is a subset of \mathbf{C} that defines the clocks to be reset. If $\lambda = \emptyset$, no reset is performed. Otherwise, the valuation for each clock $C \in \lambda$ is set to 0. For clarity, we represent a reset predicate as $C := 0$ when a single clock C needs to be reset. To denote the clock valuation identical to \mathbb{V} but with the values of clocks in λ to 0, we use $\mathbb{V}[\lambda \mapsto 0]$.

Syntax of Processes. Our session π -calculus for affine timed multiparty session types (ATMP) models timed processes interacting via affine meshed multiparty channels. It extends the calculus for affine multiparty session types (AMPST) [28] by incorporating asynchronous communication, time features, timeouts, and failure handling.¹

► **Definition 1 (Syntax).** Let $\mathbf{p}, \mathbf{q}, \mathbf{r}, \dots$ denote roles belonging to a (fixed) set \mathcal{R} ; s, s', \dots for sessions; x, y, \dots for variables; $\mathbf{m}, \mathbf{m}', \dots$ for message labels; and X, Y, \dots for process variables. The affine timed multiparty session π -calculus syntax is defined as follows:

$c, d ::= x \mid s[\mathbf{p}]$	(variable, channel with role \mathbf{p})
$P, Q ::= \mathbf{0} \mid P \parallel Q \mid (\nu s). P$	(inaction, parallel composition, restriction)
$c^n[\mathbf{q}] \oplus \mathbf{m} \langle d \rangle . P$	(timed selection towards role \mathbf{q})
$c^n[\mathbf{q}] \sum_{i \in I} \mathbf{m}_i(x_i). P_i$	(timed branching from role \mathbf{q} with $I \neq \emptyset$)
$\text{def } D \text{ in } P \mid X(\bar{c})$	(process definition, process call)
$\text{delay}(\delta) . P \mid \text{delay}(t) . P$	(time-consuming delay, deterministic delay)
$\text{timeout}[P] \mid \text{try } P \text{ catch } Q$	(timeout failure, try-catch)
$\text{cancel}(c) . P \mid \text{cerr} \mid \underline{s}$	(cancel, communication error, kill)
$s[\mathbf{p}] \blacktriangleright \sigma$	(output message queue of role \mathbf{p} in session s)
$D ::= X(\bar{x}) = P$	(declaration of process variable X)
$\sigma ::= \mathbf{q}! \mathbf{m} \langle s[\mathbf{r}] \rangle \cdot \sigma \mid \epsilon$	(message queue, non-empty or empty)

¹ To simplify, our calculus exclusively emphasises communication. Standard extensions, e.g. integers, booleans, and conditionals, are routine and independent of our formulation.

Restriction, branching, and process definitions and declarations act as binders; $\text{fc}(P)$ is the set of free channels with roles in P , $\text{fv}(P)$ is the set of free variables in P , and $\Pi_{i \in I} P_i$ is the parallel composition of processes P_i . Extensions w.r.t. AMPST calculus are highlighted. Runtime processes, generated dynamically during program execution rather than explicitly written by users, are underlined.

Our calculus comprises:

Channels c, d , being either variables x or channels with roles (a.k.a. session endpoints) $s[\mathbf{p}]$.
Standard processes as in [37, 28], including inaction $\mathbf{0}$, parallel composition $P \mid Q$, session scope restriction $(\nu s) P$, process definition **def** D **in** P , process call $X\langle \tilde{c} \rangle$, and communication error **cerr**.

Time processes that follow the program time behaviour of Fig. 2c:

- **Timed selection** (or **timed internal choice**) $c^n[\mathbf{q}] \oplus \mathbf{m}(d).P$ indicates that a message \mathbf{m} with payload d is sent to role \mathbf{q} via endpoint c , whereas **timed branching** (or **timed external choice**) $c^n[\mathbf{q}] \sum_{i \in I} \mathbf{m}_i(x_i).P_i$ waits to receive a message \mathbf{m}_i from role \mathbf{q} via endpoint c and then proceeds as P_i .

The parameter n in both timed selection and branching is a *timeout* that allows modelling different types of communication primitives: *blocking with a timeout* ($n \in \mathbb{R}_{>0}$), *blocking* ($n = \infty$), or *non-blocking* ($n = 0$). When $n \in \mathbb{R}_{\geq 0}$, the timed selection (or timed branching) process waits for up to n time units to send (or receive) a message. If the message cannot be sent (or received) within this time, the process moves into a *timeout state*, raising a *timeout failure*. If n is set to ∞ , the timed selection (or timed branching) process blocks until a message is successfully sent (or received).

In our system, we allow *send* processes to be time-consuming, enabling processes to wait before sending messages. Consider the remote data example shown in Fig. 1b. This practical scenario illustrates how a process might wait before sending a message, resulting in the possibility of send actions failing due to timeouts. It highlights the importance of timed selection, contrasting with systems like in [3] where send actions are instantaneous.

- **delay(δ)**. P represents a *time-consuming delay* action, such as method invocation or sleep. Here, δ is a clock constraint involving a *single* clock variable C , used to specify the interval for the delay. When executing **delay(δ)**. P , any time value t that satisfies the constraint δ can be consumed. Consequently, the *runtime deterministic delay* process **delay(t)**. P , arising during the execution of **delay(δ)**. P , is introduced. In **delay(t)**. P , t is a constant and a solution to δ , and P is executed after a precise delay of t time units.
- **timeout[P]** signifies that the process P has violated a time constraint, resulting in a *timeout failure*.

Failure-handling processes that adopt the AMPST approach [28]:

- **try** P **catch** Q consists of a **try** process P that is prepared to communicate with a parallel composed process, and a **catch** process Q , which becomes active in the event of a cancellation or timeout. For clarity, **try** $\mathbf{0}$ **catch** Q is not allowed within our calculus.
- **cancel(c)**. P performs the **cancellation** of other processes with channel c .
- $s \not\models \mathbf{kills}$ (terminates) all processes with session s , and is dynamically generated only at *runtime* from timeout failure or cancel processes.

Message queues: $s[\mathbf{p}] \blacktriangleright \sigma$ represents the *output message queue* of role \mathbf{p} in session s . It contains all the messages previously sent by \mathbf{p} . The queue σ can be a sequence of messages of the form $\mathbf{q}! \mathbf{m}(s[\mathbf{r}])$, where \mathbf{q} is the receiver, or ϵ , indicating an *empty* message queue. The set of *receivers in* σ , denoted as $\text{receivers}(\sigma)$, is defined in a standard way as:

$$\text{receivers}(\mathbf{q}! \mathbf{m}(s[\mathbf{r}]) \cdot \sigma') = \{\mathbf{q}\} \cup \text{receivers}(\sigma') \quad \text{receivers}(\epsilon) = \emptyset$$

[R-OUT]	$\mathbb{E} [s[\mathbf{q}]^n [\mathbf{p}] \oplus \mathbf{m} \langle s'[\mathbf{r}] \rangle . Q] \mid s[\mathbf{q}] \blacktriangleright \sigma \rightarrow Q \mid s[\mathbf{q}] \blacktriangleright \sigma \cdot \mathbf{p}! \mathbf{m} \langle s'[\mathbf{r}] \rangle \cdot \epsilon$	
[R-IN]	$\mathbb{E} [s[\mathbf{p}]^n [\mathbf{q}] \sum_{i \in I} \mathbf{m}_i(x_i) . P_i] \mid s[\mathbf{q}] \blacktriangleright \mathbf{p}! \mathbf{m}_k \langle s'[\mathbf{r}] \rangle \cdot \sigma \rightarrow P_k \{ s'[\mathbf{r}] / x_k \} \mid s[\mathbf{q}] \blacktriangleright \sigma$	($k \in I$)
[R-ERR]	$\mathbb{E} [s[\mathbf{p}]^n [\mathbf{q}] \sum_{i \in I} \mathbf{m}_i(x_i) . P_i] \mid s[\mathbf{q}] \blacktriangleright \mathbf{p}! \mathbf{m} \langle s'[\mathbf{r}] \rangle \cdot \sigma \rightarrow \mathbf{cerr}$	($\forall i \in I : \mathbf{m}_i \neq \mathbf{m}$)
[R-DET]	$\models \delta[t/C] \text{ implies } \mathbb{E}[\mathbf{delay}(\delta) . P] \rightarrow \mathbf{delay}(t) . P$	
[R-TIME]	$P \rightarrow \Psi_t(P)$	
[R-FAIL]	$\mathbf{timeout}[P] \rightarrow s \not\models$	($\exists \mathbf{r}. \mathbf{subjP}(P) = \{s[\mathbf{r}]\}$)
[R-CAN]	$\mathbb{E}[\mathbf{cancel}(s[\mathbf{p}]) . Q] \rightarrow s \not\models \mid Q$	
[R-FAILCAT]	$\mathbf{try} \mathbf{timeout}[P] \mathbf{catch} Q \rightarrow s \not\models \mid Q$	($\exists \mathbf{r}. \mathbf{subjP}(P) = \{s[\mathbf{r}]\}$)
[C-CAT]	$\mathbf{try} P \mathbf{catch} Q \mid s \not\models \rightarrow Q \mid s \not\models$	($\exists \mathbf{r}. \mathbf{subjP}(P) = \{s[\mathbf{r}]\}$)
[C-IN]	$s[\mathbf{p}]^n [\mathbf{q}] \sum_{i \in I} \mathbf{m}_i(x_i) . P_i \mid s[\mathbf{q}] \blacktriangleright \sigma \mid s \not\models$ $\rightarrow (\nu s') (P_k \{ s'[\mathbf{r}] / x_k \} \mid s' \not\models) \mid s[\mathbf{q}] \blacktriangleright \sigma \mid s \not\models$	($\mathbf{p} \notin \text{receivers}(\sigma), k \in I, s' \notin \text{fc}(P_k)$)
[C-QUEUE]	$s[\mathbf{p}] \blacktriangleright \mathbf{q}! \mathbf{m} \langle s'[\mathbf{r}] \rangle \cdot \sigma \mid s \not\models \rightarrow s[\mathbf{p}] \blacktriangleright \sigma \mid s \not\models \mid s' \not\models$	
[R-X]	$\mathbf{def} X(x_1, \dots, x_n) = P \mathbf{in} (X \langle s_1[\mathbf{p}_1], \dots, s_n[\mathbf{p}_n] \rangle \mid Q)$ $\rightarrow \mathbf{def} X(x_1, \dots, x_n) = P \mathbf{in} (P \{ s_1[\mathbf{p}_1] / x_1 \} \dots \{ s_n[\mathbf{p}_n] / x_n \} \mid Q)$	
[R-CTX]	$P \rightarrow P' \text{ implies } \mathbb{C}[P] \rightarrow \mathbb{C}[P']$	
[R-≡]	$P' \equiv P \rightarrow Q \equiv Q' \text{ implies } P' \rightarrow Q' \quad [\text{R-}\equiv\text{T}] \quad P' \equiv P \rightarrow Q \equiv Q' \text{ implies } P' \rightarrow Q'$	
[R-INS]	$P \rightarrow P' \text{ implies } P \rightarrow P'$	[R-TC] $P \rightarrow P' \text{ implies } P \rightarrow P'$
<hr/>		
$P \mid Q \equiv Q \mid P \quad (P \mid Q) \mid R \equiv P \mid (Q \mid R) \quad P \mid \mathbf{0} \equiv P \quad (\nu s) \mathbf{0} \equiv \mathbf{0} \quad (\nu s) (\nu s') P \equiv (\nu s') (\nu s) P \quad s \not\models \mid s \not\models \equiv s \not\models$		
$(\nu s) (P \mid Q) \equiv P \mid (\nu s) Q \text{ if } s \notin \text{fc}(P) \quad \mathbf{def} D \mathbf{in} \mathbf{0} \equiv \mathbf{0} \quad \mathbf{def} D \mathbf{in} (\nu s) P \equiv (\nu s) (\mathbf{def} D \mathbf{in} P) \text{ if } s \notin \text{fc}(D)$		
$\mathbf{delay}(0) . P \equiv P \quad \mathbf{def} D \mathbf{in} (P \mid Q) \equiv (\mathbf{def} D \mathbf{in} P) \mid Q \text{ if } \text{dpv}(D) \cap \text{fpv}(Q) = \emptyset$		
$(\nu s) (s[\mathbf{p}_1] \blacktriangleright \epsilon \mid \dots \mid s[\mathbf{p}_n] \blacktriangleright \epsilon) \equiv \mathbf{0} \quad \mathbf{def} D \mathbf{in} (\mathbf{def} D' \mathbf{in} P) \equiv \mathbf{def} D' \mathbf{in} (\mathbf{def} D \mathbf{in} P)$ if $(\text{dpv}(D) \cup \text{fpv}(D)) \cap \text{dpv}(D') = (\text{dpv}(D') \cup \text{fpv}(D')) \cap \text{dpv}(D) = \emptyset$		
$s[\mathbf{p}] \blacktriangleright \sigma \cdot \mathbf{q}_1! \mathbf{m}_1 \langle s_1[\mathbf{r}_1] \rangle \cdot \mathbf{q}_2! \mathbf{m}_2 \langle s_2[\mathbf{r}_2] \rangle \cdot \sigma' \equiv s[\mathbf{p}] \blacktriangleright \sigma \cdot \mathbf{q}_2! \mathbf{m}_2 \langle s_2[\mathbf{r}_2] \rangle \cdot \mathbf{q}_1! \mathbf{m}_1 \langle s_1[\mathbf{r}_1] \rangle \cdot \sigma' \quad \text{if } \mathbf{q}_1 \neq \mathbf{q}_2$		

Figure 3 Top: reduction rules for ATMP session π -calculus. Bottom: structural congruence rules for the ATMP π -calculus, where $\text{fpv}(D)$ is the set of free process variables in D , and $\text{dpv}(D)$ is the set of declared process variables in D . New rules are highlighted.

Operational Semantics. We present the operational semantics of our session π -calculus for modelling the behaviour of affine timed processes, including asynchronous communication, time progression, timeout activation, and failure handling.

► **Definition 2 (Semantics).** A **try-catch** context \mathbb{E} is defined as $\mathbb{E} ::= \mathbf{try} \mathbb{E} \mathbf{catch} P \mid []$, and a reduction context \mathbb{C} is defined as $\mathbb{C} ::= \mathbb{C} \mid P \mid (\nu s) \mathbb{C} \mid \mathbf{def} D \mathbf{in} \mathbb{C} \mid []$. The reductions \rightarrow , $\rightarrow\rightarrow$, and $\rightarrow\rightarrow\rightarrow$ are inductively defined in Fig. 3 (top), with respect to a structural congruence \equiv depicted in Fig. 3 (bottom). We write \rightarrow^* , $\rightarrow\rightarrow^*$, and $\rightarrow\rightarrow\rightarrow^*$ for their reflexive and transitive closures, respectively. $P \not\rightarrow$ (or $P \not\rightarrow\rightarrow$, $P \not\rightarrow\rightarrow\rightarrow$) means $\nexists P'$ such that $P \rightarrow P'$ (or $P \rightarrow\rightarrow P'$, $P \rightarrow\rightarrow\rightarrow P'$) is derivable. We say P has a communication error iff $\exists \mathbb{C}$ with $P = \mathbb{C}[\mathbf{cerr}]$.

We decompose the reduction rules in Fig. 3 into three relations: \rightarrow represents *instantaneous* reductions without time consumption, $\rightarrow\rightarrow$ handles time-consuming steps, and $\rightarrow\rightarrow\rightarrow$ is a general relation that can arise either from \rightarrow by [R-INS] or $\rightarrow\rightarrow$ by [R-TC]. Now let us explain the operational semantics rules for our session π -calculus.

Communication: Rules [R-OUT] and [R-IN] model asynchronous communication by queuing and dequeuing pending messages, respectively. Rule [R-ERR] is triggered by a message label mismatch, resulting in a fatal communication **error**.

Time: Rule [R-DET] specifies a deterministic delay of a specific duration t , where t is a solution to the clock constraint δ . Rule [R-TIME] incorporates a time-passing function $\Psi_t(P)$, depicted in Fig. 4, to represent time delays within a process. This *partial* function simulates a delay of time t that may occur at different parts of the process. It is *undefined* only if P is a time-consuming delay, i.e. $P = \mathbf{delay}(\delta) . P'$, or if the specified delay time t exceeds the

$$\begin{aligned}
 \Psi_t(\mathbf{0}) &= \mathbf{0} & \Psi_t(P_1 | P_2) &= \Psi_t(P_1) | \Psi_t(P_2) & \Psi_t((\nu s) P) &= (\nu s) \Psi_t(P) & \Psi_t(\text{timeout}[P]) &= \text{timeout}[P] \\
 \Psi_t(\mathbf{cerr}) &= \mathbf{cerr} & \Psi_t(\mathbf{def } D \mathbf{ in } P) &= \mathbf{def } D \mathbf{ in } \Psi_t(P) & \Psi_t(\mathbf{try } P \mathbf{ catch } Q) &= \mathbf{try } \Psi_t(P) \mathbf{ catch } \Psi_t(Q) \\
 \Psi_t(s[\mathbf{p}] \blacktriangleright \sigma) &= s[\mathbf{p}] \blacktriangleright \sigma & \Psi_t(\mathbf{delay}(\delta) . P) &= \mathbf{undefined} & \Psi_t(\mathbf{cancel}(c) . Q) &= \mathbf{cancel}(c) . \Psi_t(Q) \\
 \Psi_t(\mathbf{delay}(t') . P) &= \begin{cases} \mathbf{delay}(t' - t) . P & \text{if } t' \geq t \\ \mathbf{undefined} & \text{otherwise} \end{cases} & \Psi_t(c^\infty[\mathbf{q}] \sum_{i \in I} \mathbf{m}_i(x_i) . P_i) &= c^\infty[\mathbf{q}] \sum_{i \in I} \mathbf{m}_i(x_i) . P_i \\
 \Psi_t(c^{t'}[\mathbf{q}] \oplus \mathbf{m}\langle d \rangle . P) &= \begin{cases} c^{t'-t}[\mathbf{q}] \oplus \mathbf{m}\langle d \rangle . P & \text{if } t' \geq t \\ \mathbf{timeout}[c^{t'}[\mathbf{q}] \oplus \mathbf{m}\langle d \rangle . P] & \text{otherwise} \end{cases} & \Psi_t(c^\infty[\mathbf{q}] \oplus \mathbf{m}\langle d \rangle . P) &= c^\infty[\mathbf{q}] \oplus \mathbf{m}\langle d \rangle . P \\
 \Psi_t(s \notin) &= s \notin & \Psi_t(c^{t'}[\mathbf{q}] \sum_{i \in I} \mathbf{m}_i(x_i) . P_i) &= \begin{cases} c^{t'-t}[\mathbf{q}] \sum_{i \in I} \mathbf{m}_i(x_i) . P_i & \text{if } t' \geq t \\ \mathbf{timeout}[c^{t'}[\mathbf{q}] \sum_{i \in I} \mathbf{m}_i(x_i) . P_i] & \text{otherwise} \end{cases}
 \end{aligned}$$

Figure 4 Time-passing function $\Psi_t(P)$.

duration of a runtime deterministic delay, i.e. $P = \mathbf{delay}(t') . P'$ with $t > t'$. The latter case arises because deterministic delays must always adhere to their specified durations, e.g. if a program is instructed to sleep for 5 time units, it must strictly follow this duration.

Notably, $\Psi_t(P)$ acts as the *only mechanism* for triggering a timeout failure $\text{timeout}[P]$, resulting from a timed selection or branching. Such a timeout failure occurs when $\Psi_t(P)$ is defined, and the specified delay t exceeds a *deadline* set within P .

Cancellation: Rules [C-IN] and [C-QUEUE] model the process cancellations. [C-IN] is triggered only when there are no messages in the queue that can be received from \mathbf{q} via the endpoint $s[\mathbf{p}]$. Cancellation of a timed selection is expected to eventually occur via [C-QUEUE]; therefore, there is no specific rule dedicated to it. Similarly, in our implementation, the timed selection is not directly cancelled either.

Rules [R-CAN] and [C-CAT], adapted from [28], state cancellations from other parties. [R-CAN] facilitates cancellation and generates a kill process, while [C-CAT] transitions to the **catch** process Q due to the termination of session s , where the **try** process P is communicating on s . Therefore, the set of *subjects* of process P , denoted as $\text{subjP}(P)$, is included in the side condition of [C-CAT] to ensure that P has a prefix at s , as defined below:

$$\begin{aligned}
 \text{subjP}(\mathbf{0}) &= \text{subjP}(\mathbf{cerr}) = \emptyset & \text{subjP}(P | Q) &= \text{subjP}(P) \cup \text{subjP}(Q) & \text{subjP}(s[\mathbf{p}] \blacktriangleright \sigma) &= \{s[\mathbf{p}]^Q\} \\
 \text{subjP}((\nu s) P) &= \text{subjP}(P) \setminus \{s[\mathbf{p}_i]\}_{i \in I} \cup \{s[\mathbf{p}_i]^Q\}_{i \in I} \\
 \text{subjP}(\mathbf{def } X(\widetilde{x}) = P \mathbf{ in } Q) &= \text{subjP}(Q) \cup \text{subjP}(P) \setminus \{\widetilde{x}\} \text{ with } \text{subjP}(X\langle \widetilde{c} \rangle) = \text{subjP}(P\{\widetilde{c}/\widetilde{x}\}) \\
 \text{subjP}(c^n[\mathbf{q}] \oplus \mathbf{m}\langle d \rangle . P) &= \text{subjP}(c^n[\mathbf{q}] \sum_{i \in I} \mathbf{m}_i(x_i) . P_i) = \text{subjP}(\mathbf{cancel}(c) . P) = \{c\} \\
 \text{subjP}(\mathbf{delay}(\delta) . P) &= \text{subjP}(\mathbf{delay}(t) . P) = \text{subjP}(\mathbf{try } P \mathbf{ catch } Q) = \text{subjP}(\text{timeout}[P]) = \text{subjP}(P)
 \end{aligned}$$

Subjects of processes determine sessions that may need cancellation, a crucial aspect for handling failed or cancelled processes properly. In our definition, subjects not only denote the endpoints via which processes start interacting but also indicate whether they are used for message queue processes. Specifically, an endpoint $s[\mathbf{p}]$ annotated with Q signifies its use in a queue process. This additional annotation, and thus the distinction it implies, is pivotal in formulating the typing rule for the **try-catch** process, as discussed later in § 4.4, where we rely on subjects to exclude queue processes within any **try** construct.

Timeout Handling: Rules [R-FAIL] and [R-FAILCAT] address time failures. In the event of a timeout, a killing process is generated. Moreover, in [R-FAILCAT], the **catch** process Q is triggered. To identify the session requiring termination, the set of subjects of the failure process $\text{timeout}[P]$ is considered in both rules as a side condition. Note that a timeout arises exclusively from timed selection or branching. Therefore, the subject set of $\text{timeout}[P]$ must contain a *single* endpoint devoid of Q , indicating the generation of only one killing process.

Standard: Rules [R-X], [R-CTX], and [R-≡] are standard [37, 28]. [R-X] expands process definitions when invoked; [R-CTX] and [R-≡] allow processes to reduce under reduction contexts

and through structural congruence, respectively. Rule $[R \equiv T]$ introduces a timed variant of $[R \equiv]$, enabling time-consuming reductions via structural congruence.

Congruence: As shown in Fig. 3 (bottom), we introduce additional congruence rules related to queues, delays, and process killings, alongside standard rules from [37]. Specifically, two rules are proposed for queues: the first addresses the *garbage* collection of queues that are not referenced by any process, while the second rearranges messages with different receivers. The rule for delays states that adding a delay of zero time units has no effect on the process execution. The rule regarding process killings eliminates duplicate kills.

► **Example 3.** Consider the processes: $P_1 = s[\text{Sat}]^{0.4}[\text{Ser}] \oplus \text{Data.0}$, $P_2 = s[\text{Ser}]^{0.3}[\text{Sat}]\text{Data.0}$, and $P_3 = s[\text{Sat}] \blacktriangleright \epsilon$. Rule $[C\text{-CAT}]$ can be applied to $\text{try } P_1 \text{ catch } Q \mid s \not\in$, as $\text{subjP}(P_1) = \{s[\text{Sat}]\}$ satisfies its side condition. However, neither $\text{timeout}[P_1 \mid P_2]$ nor $\text{timeout}[P_3]$ can generate the killing process $s \not\in$, as $\text{subjP}(P_1 \mid P_2) = \{s[\text{Sat}], s[\text{Ser}]\}$, whereas $\text{subjP}(P_3) = \{s[\text{Sat}]^0\}$.

► **Example 4.** Processes Q_{Sen} , Q_{Sat} , and Q_{Ser} interact on a session s :

$$\begin{aligned} Q_{\text{Sen}} &= \text{delay}(C_{\text{Sen}} = 6.5) \cdot Q'_{\text{Sen}} \mid s[\text{Sen}] \blacktriangleright \epsilon \text{ where } Q'_{\text{Sen}} = \text{try } s[\text{Sen}]^{0.3}[\text{Sat}] \oplus \text{Data catch cancel}(s[\text{Sen}]) \\ Q_{\text{Sat}} &= \text{delay}(C_{\text{Sat}} = 6) \cdot Q'_{\text{Sat}} \mid s[\text{Sat}] \blacktriangleright \epsilon \text{ where } Q'_{\text{Sat}} = s[\text{Sat}]^{0.2}[\text{Sen}] \sum \left\{ \begin{array}{l} \text{Data.} s[\text{Sat}]^{0.3}[\text{Ser}] \oplus \text{Data} \\ \text{fail.} s[\text{Sat}]^{0.4}[\text{Ser}] \oplus \text{fatal} \end{array} \right\} \\ Q_{\text{Ser}} &= \text{delay}(C_{\text{Ser}} = 6) \cdot Q'_{\text{Ser}} \mid s[\text{Ser}] \blacktriangleright \epsilon \text{ where } Q'_{\text{Ser}} = s[\text{Ser}]^{0.8}[\text{Sat}] \sum \{\text{Data, fatal}\} \end{aligned}$$

Process Q_{Sen} delays for exactly 6.5 time units before executing process Q'_{Sen} . Here, Q'_{Sen} attempts to use $s[\text{Sen}]$ to send **Data** to **Sat** within 0.3 time units. If the attempt fails, the cancellation of $s[\text{Sen}]$ is triggered. Process Q_{Sat} waits for precisely 6 time units before using $s[\text{Sat}]$ to receive either **Data** or **fail** from **Sen** within 0.2 time units; subsequently, in the first case, it uses $s[\text{Sat}]$ to send **Data** to **Ser** within 0.3 time units, while in the latter, it uses $s[\text{Sat}]$ to send **fail** to **Ser** within 0.4 time units. Similarly, process Q_{Ser} waits 6 time units before using $s[\text{Ser}]$ to receive either **Data** or **fatal** from **Sat** within 0.8 time units.

In Q_{Sen} , $s[\text{Sen}]$ can only start sending **Data** to **Sat** after 6.5 time units, whereas in Q_{Sat} , $s[\text{Sat}]$ must receive the message from **Sen** within 0.2 time units after a 6-time unit delay. Consequently, $s[\text{Sat}]$ fails to receive the message from **Sen** within the specified interval, resulting in a timeout failure, i.e.

$$\begin{aligned} Q_{\text{Sen}} \mid Q_{\text{Sat}} \mid Q_{\text{Ser}} &\rightsquigarrow \text{delay}(6.5) \cdot Q'_{\text{Sen}} \mid s[\text{Sen}] \blacktriangleright \epsilon \mid \text{delay}(6) \cdot Q'_{\text{Sat}} \mid s[\text{Sat}] \blacktriangleright \epsilon \mid \text{delay}(6) \cdot Q'_{\text{Ser}} \mid s[\text{Ser}] \blacktriangleright \epsilon \\ &\rightarrow \Psi_{6.5}(\text{delay}(6.5) \cdot Q'_{\text{Sen}} \mid s[\text{Sen}] \blacktriangleright \epsilon \mid \text{delay}(6) \cdot Q'_{\text{Sat}} \mid s[\text{Sat}] \blacktriangleright \epsilon \mid \text{delay}(6) \cdot Q'_{\text{Ser}} \mid s[\text{Ser}] \blacktriangleright \epsilon) \\ &\equiv Q'_{\text{Sen}} \mid s[\text{Sen}] \blacktriangleright \epsilon \mid \text{timeout}[Q'_{\text{Sat}}] \mid s[\text{Sat}] \blacktriangleright \epsilon \mid \Psi_{0.5}(Q'_{\text{Ser}}) \mid s[\text{Ser}] \blacktriangleright \epsilon \end{aligned}$$

Therefore, the kill process $s \not\in$ is generated from $\text{timeout}[Q'_{\text{Sat}}]$, successfully terminating the process $Q_{\text{Sen}} \mid Q_{\text{Sat}} \mid Q_{\text{Ser}}$ by the following reductions:

$$\begin{aligned} &Q'_{\text{Sen}} \mid s[\text{Sen}] \blacktriangleright \epsilon \mid \text{timeout}[Q'_{\text{Sat}}] \mid s[\text{Sat}] \blacktriangleright \epsilon \mid \Psi_{0.5}(Q'_{\text{Ser}}) \mid s[\text{Ser}] \blacktriangleright \epsilon \\ &\rightsquigarrow Q'_{\text{Sen}} \mid s[\text{Sen}] \blacktriangleright \epsilon \mid s \not\in \mid s[\text{Sat}] \blacktriangleright \epsilon \mid \Psi_{0.5}(Q'_{\text{Ser}}) \mid s[\text{Ser}] \blacktriangleright \epsilon \\ &\rightsquigarrow \text{cancel}(s[\text{Sen}]) \mid s[\text{Sen}] \blacktriangleright \epsilon \mid s \not\in \mid s[\text{Sat}] \blacktriangleright \epsilon \mid 0 \mid s[\text{Ser}] \blacktriangleright \epsilon \\ &\rightsquigarrow s \not\in \mid 0 \mid s[\text{Sen}] \blacktriangleright \epsilon \mid s \not\in \mid s[\text{Sat}] \blacktriangleright \epsilon \mid 0 \mid s[\text{Ser}] \blacktriangleright \epsilon \equiv 0 \mid s \not\in \end{aligned}$$

4 Affine Timed Multiparty Session Type System

In this section, we introduce our affine timed multiparty session type system. We begin by exploring the types used in ATMP, as well as subtyping and projection, in § 4.1. We furnish a Labelled Transition System (LTS) semantics for typing environments (collections of timed local types and queue types) in § 4.2, and timed global types in § 4.3, illustrating their relationship with Thms. 13 and 14. Furthermore, we present a type system for our ATMP session π -calculus in § 4.4. Finally, we show the main properties of the type system: *subject reduction* (Thm. 17), *session fidelity* (Thm. 21), and *deadlock-freedom* (Thm. 24), in § 4.5.

$S ::= (\delta, \textcolor{blue}{T})$	sort
$G ::= \textcolor{red}{p} \rightarrow \textcolor{brown}{q}; \{\textcolor{brown}{m}_i(S_i)\{\delta_{O_i}, \lambda_{O_i}, \delta_{I_i}, \lambda_{I_i}\}.G_i\}_{i \in I}$	transmission
$\textcolor{brown}{p} \rightsquigarrow \textcolor{brown}{q}; j \{\textcolor{brown}{m}_i(S_i)\{\delta_{O_i}, \lambda_{O_i}, \delta_{I_i}, \lambda_{I_i}\}.G_i\}_{i \in I} \quad (j \in I)$	transmission en route
$\mu t.G \quad \quad \mathbf{t} \quad \quad \mathbf{end}$	recursion, type variable, termination
$T ::= \textcolor{blue}{p} \& \{\textcolor{brown}{m}_i(S_i)\{\delta_i, \lambda_i\}.T_i\}_{i \in I} \quad \quad \textcolor{blue}{p} \oplus \{\textcolor{brown}{m}_i(S_i)\{\delta_i, \lambda_i\}.T_i\}_{i \in I}$	external choice, internal choice
$\mu t.T \quad \quad \mathbf{t} \quad \quad \mathbf{end}$	recursion, type variable, termination
$\mathcal{M} ::= \textcolor{blue}{p}! \textcolor{brown}{m}(S) \cdot \mathcal{M} \quad \quad \emptyset$	queue types

■ **Figure 5** Syntax of timed global types, timed local types, and queue types.

4.1 Timed Multiparty Session Types

Affine session frameworks keep the original system’s type-level syntax intact, requiring no changes. To introduce affine timed asynchronous multiparty session types, we simply need to augment global and local types with clock constraints and resets introduced in §3 to derive *timed global and local types*. The syntax of types used in this paper is presented in Fig. 5. As usual, all types are required to be closed and have guarded recursion variables.

Sorts. Sorts are ranged over S, S', S_i, \dots , and facilitate the delegation of the remaining behaviour $\textcolor{blue}{T}$ to the receiver, who can execute it under any clock assignment satisfying δ .

Timed Global Types. Timed global types are ranged over G, G', G_i, \dots , and describe an *overview* of the behaviour for all roles ($\textcolor{red}{p}, \textcolor{brown}{q}, \textcolor{brown}{s}, \textcolor{brown}{t}, \dots$) belonging to a (fixed) set \mathcal{R} . The set of roles in a timed global type G is denoted as $\text{roles}(G)$, while the set of its free variables as $\text{fv}(G)$.

A transmission $\textcolor{red}{p} \rightarrow \textcolor{brown}{q}; \{\textcolor{brown}{m}_i(\textcolor{blue}{S}_i)\{\delta_{O_i}, \lambda_{O_i}, \delta_{I_i}, \lambda_{I_i}\}.G_i\}_{i \in I}$ represents a message sent from role $\textcolor{red}{p}$ to role $\textcolor{brown}{q}$, with labels $\textcolor{brown}{m}_i$, payload types $\textcolor{blue}{S}_i$ (which are sorts), and continuations G_i , where i is taken from an index set I , and $\textcolor{brown}{m}_i$ taken from a fixed set of all labels \mathcal{M} . Each branch is associated with a *time assertion* consisting of four components: δ_{O_i} and λ_{O_i} for the output (sending) action, and δ_{I_i} and λ_{I_i} for the input (receiving) action. These components specify the clock constraint and reset predicate for the respective actions. A message can be sent (or received) at any time satisfying the guard δ_{O_i} (or δ_{I_i}), and the clocks in λ_{O_i} (or λ_{I_i}) are reset upon sending (or receiving). In addition to the standard requirements for global types as in [11], we impose a condition from [4], stating that sets of clocks “owned” by different roles, i.e. those that can be read and reset, must be pairwise disjoint. Furthermore, the clock constraint and reset predicate of an output or input action performed by a role are defined only over the clocks owned by that role.

A transmission en route $\textcolor{red}{p} \rightsquigarrow \textcolor{brown}{q}; j \{\textcolor{brown}{m}_i(\textcolor{blue}{S}_i)\{\delta_{O_i}, \lambda_{O_i}, \delta_{I_i}, \lambda_{I_i}\}.G_i\}_{i \in I} \quad (j \in I)$ is a *runtime* construct to represent a message $\textcolor{brown}{m}_j$ sent by $\textcolor{red}{p}$, and yet to be received by $\textcolor{brown}{q}$. Recursion $\mu t.G$ and termination \mathbf{end} (omitted where unambiguous) are standard [11]. Note that contractive requirements [34, §21.8], i.e. ensuring that each recursion variable \mathbf{t} is bound within a $\mu t \dots$ and is guarded, are applied in recursive types.

Timed Local Types. Timed local types (or *timed session types*) are ranged over $\textcolor{blue}{T}, \textcolor{blue}{U}, \textcolor{blue}{T}', \textcolor{blue}{U}'$, $\textcolor{blue}{T}_i, \textcolor{blue}{U}_i, \dots$, and describe the behaviour of a *single* role. An *internal choice (selection)* $\textcolor{blue}{p} \oplus \{\textcolor{brown}{m}_i(\textcolor{blue}{S}_i)\{\delta_i, \lambda_i\}.T_i\}_{i \in I}$ (or *external choice (branching)* $\textcolor{blue}{p} \& \{\textcolor{brown}{m}_i(\textcolor{blue}{S}_i)\{\delta_i, \lambda_i\}.T_i\}_{i \in I}$) states that the *current* role is to *send to* (or *receive from*) the role $\textcolor{red}{p}$ when δ_i is satisfied, followed by resetting the clocks in λ_i . *Recursive* and *termination* types are defined similarly to timed global types. The requirements for the index set, labels, clock constraints, and reset predicates in timed local types mirror those in timed global types.

Queue Types. Queue Types are ranged over $\mathcal{M}, \mathcal{M}', \mathcal{M}_i, \dots$, and represent (possibly empty) sequences of *message types* $p!m(S)$ having receiver p , label m , and payload type S (omitted when $S=(\delta, \text{end})$). As interactions in our formalisation are asynchronous, queue types are used to capture the states in which messages are in transit. We adopt the notation receivers(\cdot) from §3 to denote the set of *receivers* in \mathcal{M} as receivers(\mathcal{M}) as well, with a similar definition.

Subtyping. We introduce a subtyping relation \leqslant on timed local types in Def. 5, based on the standard behaviour-preserving subtyping [37]. This relation indicates that a smaller type entails fewer external choices but more internal choices.

► **Definition 5** (Subtyping). *The subtyping relation \leqslant is coinductively defined:*

$$\begin{array}{c} \frac{\forall i \in I \quad S'_i \leqslant S_i \quad \delta_i = \delta'_i \quad \lambda_i = \lambda'_i \quad T_i \leqslant T'_i}{p \oplus \{m_i(S_i)\{\delta_i, \lambda_i\}.T_i\}_{i \in I \cup J} \leqslant p \oplus \{m_i(S'_i)\{\delta'_i, \lambda'_i\}.T'_i\}_{i \in I}} \text{ [SUB-}\oplus\text{]} \\ \frac{\forall i \in I \quad S_i \leqslant S'_i \quad \delta_i = \delta'_i \quad \lambda_i = \lambda'_i \quad T_i \leqslant T'_i}{p \& \{m_i(S_i)\{\delta_i, \lambda_i\}.T_i\}_{i \in I} \leqslant p \& \{m_i(S'_i)\{\delta'_i, \lambda'_i\}.T'_i\}_{i \in I \cup J}} \text{ [SUB-}\&\text{]} \quad \frac{}{\text{end} \leqslant \text{end}} \text{ [SUB-end]} \\ \frac{T \leqslant T'}{\langle \delta, T \rangle \leqslant \langle \delta, T' \rangle} \text{ [SUB-S]} \quad \frac{T \{\mu t.T/t\} \leqslant T'}{\mu t.T \leqslant T'} \text{ [SUB-}\mu\text{L]} \quad \frac{T \leqslant T' \{\mu t.T'/t\}}{T \leqslant \mu t.T'} \text{ [SUB-}\mu\text{R]} \end{array}$$

Projection. Projection of a timed global type G onto a role p yields a timed local type. Our definition of projection in Def. 6 is mostly standard [37], with the addition of projecting time assertions onto the sender and receiver, respectively.

► **Definition 6** (Projection). *The projection of a timed global type G onto a role p , written as $G \upharpoonright p$, is:*

$$\begin{aligned} (q \rightarrow r : \{m_i(S_i)\{\delta_{O_i}, \lambda_{O_i}, \delta_{I_i}, \lambda_{I_i}\}.G_i\}_{i \in I}) \upharpoonright p &= \begin{cases} r \oplus \{m_i(S_i)\{\delta_{O_i}, \lambda_{O_i}\}.(G_i \upharpoonright p)\}_{i \in I} & \text{if } p = q \\ q \& \{m_i(S_i)\{\delta_{I_i}, \lambda_{I_i}\}.(G_i \upharpoonright p)\}_{i \in I} & \text{if } p = r \\ \prod_{i \in I} G_i \upharpoonright p & \text{otherwise} \end{cases} \\ (q \rightsquigarrow r : j \{m_i(S_i)\{\delta_{O_i}, \lambda_{O_i}, \delta_{I_i}, \lambda_{I_i}\}.G_i\}_{i \in I}) \upharpoonright p &= \begin{cases} G_j \upharpoonright p & \text{if } p = q \\ q \& \{m_i(S_i)\{\delta_{I_i}, \lambda_{I_i}\}.(G_i \upharpoonright p)\}_{i \in I} & \text{if } p = r \\ \prod_{i \in I} G_i \upharpoonright p & \text{otherwise} \end{cases} \\ (\mu t.G) \upharpoonright p &= \begin{cases} \mu t.(G \upharpoonright p) & \text{if } p \in \text{roles}(G) \text{ or } \text{fv}(\mu t.G) \neq \emptyset \\ \text{end} & \text{otherwise} \end{cases} \quad \begin{cases} t \upharpoonright p = t & \\ \text{end} \upharpoonright p = \text{end} & \end{cases} \end{aligned}$$

where \prod is the merge operator for timed session types:

$$\begin{aligned} p \& \{m_i(S_i)\{\delta_i, \lambda_i\}.T_i\}_{i \in I} \sqcap p \& \{m_j(S'_j)\{\delta'_j, \lambda'_j\}.T'_j\}_{j \in J} &= \\ p \& \{m_k(S_k)\{\delta_k, \lambda_k\}.(T_k \sqcap T'_k)\}_{k \in I \cap J} \& p \& \{m_i(S_i)\{\delta_i, \lambda_i\}.T_i\}_{i \in I \setminus J} \& p \& \{m_j(S'_j)\{\delta'_j, \lambda'_j\}.T'_j\}_{j \in J \setminus I} \\ p \oplus \{m_i(S_i)\{\delta_i, \lambda_i\}.T_i\}_{i \in I} \sqcap p \oplus \{m_i(S_i)\{\delta_i, \lambda_i\}.T'_i\}_{i \in I} &= p \oplus \{m_i(S_i)\{\delta_i, \lambda_i\}.(T_i \sqcap T'_i)\}_{i \in I} \\ \mu t.T \sqcap \mu t.U &= \mu t.(T \sqcap U) \quad t \sqcap t = t \quad \text{end} \sqcap \text{end} = \text{end} \end{aligned}$$

► **Example 7.** Take the timed global type G , and timed local types T_{Sat} and T_{Ser} from §2.1. Consider a timed global type G_{data} , derived from remote data (Fig. 1b) as well, representing data transmission from **Sen** to **Ser** via **Sat**:

$$G_{\text{data}} = \text{Sen} \rightarrow \text{Sat} : \{\text{Data}\{6 \leq C_{\text{Sen}} \leq 7, C_{\text{Sen}} := 0, 6 \leq C_{\text{Sat}} \leq 7, \emptyset\}.G\}$$

which can be projected onto roles **Sen**, **Sat**, and **Ser**, respectively, as:

$$\begin{aligned} G_{\text{data}} \upharpoonright \text{Sen} &= \text{Sat} \oplus \text{Data}\{6 \leq C_{\text{Sen}} \leq 7, C_{\text{Sen}} := 0\}.end \quad G_{\text{data}} \upharpoonright \text{Ser} = G \upharpoonright \text{Ser} = T_{\text{Ser}} \\ G_{\text{data}} \upharpoonright \text{Sat} &= \text{Sen} \& \text{Data}\{6 \leq C_{\text{Sat}} \leq 7, \emptyset\}.G \upharpoonright \text{Sat} = \text{Sen} \& \text{Data}\{6 \leq C_{\text{Sat}} \leq 7, \emptyset\}.T_{\text{Sat}} \end{aligned}$$

$$\begin{array}{c}
\text{Congruence (top)} \\
\frac{\text{p} \neq \text{q}}{(\mathbb{V}, T) \equiv (\mathbb{V}, T)} \quad \frac{\text{p}!m_1(S_1) \cdot \text{q}!m_2(S_2) \cdot \mathcal{M} \equiv \text{q}!m_2(S_2) \cdot \text{p}!m_1(S_1) \cdot \mathcal{M}}{\mathcal{M} \equiv \mathcal{M}' \quad (\mathbb{V}, T) \equiv (\mathbb{V}, T')} \\
\frac{\mathcal{M} \equiv \mathcal{M}'}{\mathcal{M}; (\mathbb{V}, T) \equiv \mathcal{M}'; (\mathbb{V}, T')} \\
\text{Subtyping (bottom)} \\
\frac{\text{p}!m(S) \cdot \mathcal{O} \cdot \mathcal{M} \equiv \mathcal{O} \cdot \text{p}!m(S) \cdot \mathcal{M}}{\mathcal{O} \leqslant \mathcal{O}} \quad \frac{\text{S}' \leqslant S \quad \mathcal{M} \leqslant \mathcal{M}'}{\text{q}!m(S) \cdot \mathcal{M} \leqslant \text{q}!m(S') \cdot \mathcal{M}'} \quad \frac{T \leqslant T'}{(\mathbb{V}, T) \leqslant (\mathbb{V}, T')} \quad \frac{\mathcal{M} \leqslant \mathcal{M}'}{\mathcal{M}; (\mathbb{V}, T) \leqslant \mathcal{M}'; (\mathbb{V}, T')}
\end{array}$$

Figure 6 Congruence (top) and subtyping (bottom) rules for timed-session/queue types.

4.2 Typing Environments

To reflect the behaviour of timed global types (§ 4.3), present a typing system for our session π -calculus (§ 4.4), and introduce type-level properties (§ 4.5), we formalise *typing environments* in Def. 8, followed by their Labelled Transition System (LTS) semantics in Def. 9.

► **Definition 8** (Typing Environments). *The typing environments Θ and Γ are defined as:*

$$\Theta ::= \emptyset \mid \Theta, X:(\mathbb{V}_1, T_1), \dots, (\mathbb{V}_n, T_n) \quad \Gamma ::= \emptyset \mid \Gamma, x:(\mathbb{V}, T) \mid \Gamma, s[p]:\tau$$

where τ is a timed-session/queue type: $\tau ::= (\mathbb{V}, T) \mid \mathcal{M} \mid \mathcal{M}; (\mathbb{V}, T)$, i.e. either a timed session type, a queue type, or a combination.

The environment composition Γ_1, Γ_2 is defined iff $\forall c \in \text{dom}(\Gamma_1) \cap \text{dom}(\Gamma_2) : \Gamma_i(c) = \mathcal{M}$ and $\Gamma_j(c) = (\mathbb{V}, T)$ with $i, j \in \{1, 2\}$, and for all such c , we posit $(\Gamma_1, \Gamma_2)(c) = \mathcal{M}; (\mathbb{V}, T)$.

We write $\text{dom}(\Gamma) = \{s\}$ iff for any $c \in \text{dom}(\Gamma)$, there is p such that $c = s[p]$ (i.e. Γ only contains session s). We write $s \notin \Gamma$ iff $\forall p : s[p] \notin \text{dom}(\Gamma)$ (i.e. session s does not occur in Γ). We write Γ_s iff $\text{dom}(\Gamma_s) = \{s\}$, $\text{dom}(\Gamma_s) \subseteq \text{dom}(\Gamma)$, and $\forall s[p] \in \text{dom}(\Gamma) : \Gamma(s[p]) = \Gamma_s(s[p])$ (i.e. restriction of Γ to session s). We denote updates as $\Gamma[c \mapsto \tau] : \Gamma[c \mapsto \tau](c) = \tau$ and $\Gamma[c \mapsto \tau](c') = \Gamma(c')$ (where $c \neq c'$).

Congruence and subtyping are imposed on typing environments: $\Gamma \equiv \Gamma'$ (resp. $\Gamma \leqslant \Gamma'$) iff $\text{dom}(\Gamma) = \text{dom}(\Gamma')$ and $\forall c \in \text{dom}(\Gamma) : \Gamma(c) \equiv \Gamma'(c)$ (resp. $\Gamma(c) \leqslant \Gamma'(c)$), incorporating additional congruence and subtyping rules for time-session/queue types, as depicted in Fig. 6.

In Def. 8, the typing environment Θ maps process variables to n -tuples of timed session types, while Γ maps variables to timed session types, and channels with roles to timed-session/queue types. Note that in our typing environments, timed session types are annotated with clock valuations, denoted as (\mathbb{V}, T) . This enables us to capture timing information within the type system, facilitating the tracking of the (virtual) time at which the next action can be validated during the execution of a process.

The congruence relation \equiv for timed-session/queue types is inductively defined as in Fig. 6 (top), reordering queued messages with different receivers. Subtyping for timed-session/queue types extends Def. 5 with rules in Fig. 6 (bottom): particularly, rule [SUB- \mathcal{M}] states that a sequence of queued message types is a subtype of another if messages in the same position have identical receivers and labels, and their payload sorts are related by subtyping.

► **Definition 9** (Typing Environment Reduction). *Let α be a transition label of the form $s:p!q:m$, $s:p,q:m$, or t . The typing environment transition $\xrightarrow{\alpha}$ is inductively defined by the rules in Fig. 7 (top). We write $\Gamma \xrightarrow{\alpha}$ iff $\Gamma \xrightarrow{\alpha} \Gamma'$ for some Γ' . We define two reductions $\Gamma \xrightarrow{s} \Gamma'$ (where s is a session) and $\Gamma \xrightarrow{} \Gamma'$ as follows:*

- $\Gamma \xrightarrow{s} \Gamma'$ holds iff $\Gamma \xrightarrow{\alpha} \Gamma'$ with $\alpha \in \{s:p!q:m, s:p,q:m, t \mid p, q \in \mathcal{R}\}$ (where \mathcal{R} is the set of all roles). We write $\Gamma \xrightarrow{} \Gamma$ iff $\Gamma \xrightarrow{s} \Gamma'$ for some Γ' , and $\xrightarrow{*}$ as the reflexive and transitive closure of \xrightarrow{s} ;
- $\Gamma \xrightarrow{} \Gamma'$ holds iff $\Gamma \xrightarrow{s} \Gamma'$ for some s . We write $\Gamma \xrightarrow{} \Gamma$ iff $\Gamma \xrightarrow{} \Gamma'$ for some Γ' , and $\xrightarrow{*}$ as the reflexive and transitive closure of $\xrightarrow{}$.

$$\begin{array}{c}
\frac{\Gamma, s[\mathbf{p}]:(\mathbb{V}, T\{\mu t.T/\mathbf{t}\}) \xrightarrow{\alpha} \Gamma'}{\Gamma, s[\mathbf{p}]:(\mathbb{V}, \mu t.T) \xrightarrow{\alpha} \Gamma'} \text{ [Γ-μ]} \quad \frac{\Gamma \xrightarrow{\alpha} \Gamma' \quad \alpha \neq t}{\Gamma, x:(\mathbb{V}, T) \xrightarrow{\alpha} \Gamma', x:(\mathbb{V}, T)} \text{ [Γ-,x]} \quad \frac{\Gamma \xrightarrow{\alpha} \Gamma' \quad \alpha \neq t}{\Gamma, s[\mathbf{p}]:\tau \xrightarrow{\alpha} \Gamma', s[\mathbf{p}]:\tau} \text{ [Γ-,τ]} \\
\frac{k \in I \quad \mathbb{V} \models \delta_k}{s[\mathbf{p}]:\mathcal{M}; (\mathbb{V}, \mathbf{q} \oplus \{\mathbf{m}_i(S_i)\{\delta_i, \lambda_i\}.T_i\}_{i \in I}) \xrightarrow{s:\mathbf{p}!\mathbf{q}:\mathbf{m}_k} s[\mathbf{p}]:\mathcal{M} \cdot \mathbf{q}!\mathbf{m}_k(S_k) \cdot \emptyset; (\mathbb{V}[\lambda_k \mapsto 0], T_k)} \text{ [Γ-⊕]} \\
\frac{k \in I \quad \mathbb{V} \models \delta_k \quad S_k \leq S'_k}{s[\mathbf{p}]:\mathbf{q}!\mathbf{m}_k(S_k) \cdot \mathcal{M}, s[\mathbf{q}]:(\mathbb{V}, \mathbf{p} \& \{\mathbf{m}_i(S'_i)\{\delta_i, \lambda_i\}.T_i\}_{i \in I}) \xrightarrow{s:\mathbf{q},\mathbf{p}:\mathbf{m}_k} s[\mathbf{p}]:\mathcal{M}, s[\mathbf{q}]:(\mathbb{V}[\lambda_k \mapsto 0], T_k)} \text{ [Γ-&]} \\
\frac{c:(\mathbb{V}, T) \xrightarrow{t} c:(\mathbb{V} + t, T) \text{ [Γ-Ts]} \quad s[\mathbf{p}]:\mathcal{M} \xrightarrow{t} s[\mathbf{p}]:\mathcal{M} \text{ [Γ-TQ]} \quad s[\mathbf{p}]:\mathcal{M}; (\mathbb{V}, T) \xrightarrow{t} s[\mathbf{p}]:\mathcal{M}; (\mathbb{V} + t, T) \text{ [Γ-Tc]}}{\frac{\Gamma_1 \xrightarrow{t} \Gamma'_1 \quad \Gamma_2 \xrightarrow{t} \Gamma'_2}{\Gamma_1, \Gamma_2 \xrightarrow{t} \Gamma'_1, \Gamma'_2} \text{ [Γ-,T]} \quad \frac{\Gamma \equiv \Gamma_1 \quad \Gamma_1 \xrightarrow{\alpha} \Gamma'_1 \quad \Gamma'_1 \equiv \Gamma'}{\Gamma \xrightarrow{\alpha} \Gamma'} \text{ [Γ-STRUCT]}} \\
\hline
\frac{\langle \mathbb{V}; G \rangle \xrightarrow{t} \langle \mathbb{V} + t; G \rangle \text{ [GR-T]} \quad \frac{\langle \mathbb{V}; G\{\mu t.G/\mathbf{t}\} \rangle \xrightarrow{\alpha} \langle \mathbb{V}'; G' \rangle \quad \langle \mathbb{V}; \mu t.G \rangle \xrightarrow{\alpha} \langle \mathbb{V}'; G' \rangle}{j \in I \quad \mathbb{V} \models \delta_{O,j} \quad \mathbb{V}' = \mathbb{V}[\lambda_{O,j} \mapsto 0]} \text{ [GR-μ]} \quad \frac{j \in I \quad \mathbb{V} \models \delta_{I,j} \quad \mathbb{V}' = \mathbb{V}[\lambda_{I,j} \mapsto 0]}{\langle \mathbb{V}; \mathbf{p} \rightarrow \mathbf{q}: \{\mathbf{m}_i(S_i)\{\mathcal{A}_i\}.G'_i\}_{i \in I} \xrightarrow{s:\mathbf{p}!\mathbf{q}:\mathbf{m}_j} \langle \mathbb{V}'; \mathbf{p} \rightsquigarrow \mathbf{q}: j \{\mathbf{m}_i(S_i)\{\mathcal{A}_i\}.G'_i\}_{i \in I} \rangle} \text{ [GR-⊕]} \\
\frac{\langle \mathbb{V}; \mathbf{p} \rightsquigarrow \mathbf{q}: j \{\mathbf{m}_i(S_i)\{\mathcal{A}_i\}.G'_i\}_{i \in I} \xrightarrow{s:\mathbf{q},\mathbf{p}:\mathbf{m}_j} \langle \mathbb{V}'; G'_j \rangle}{\forall i \in I : \langle \mathbb{V}; G'_i \rangle \xrightarrow{\alpha} \langle \mathbb{V}'; G''_i \rangle \quad \mathbf{p}, \mathbf{q} \notin \text{subject}(\alpha) \quad \alpha \neq t} \text{ [GR-CTX-I]} \\
\frac{\forall i \in I : \langle \mathbb{V}; G'_i \rangle \xrightarrow{\alpha} \langle \mathbb{V}'; G''_i \rangle \quad \mathbf{q} \notin \text{subject}(\alpha) \quad \alpha \neq t}{\langle \mathbb{V}; \mathbf{p} \rightsquigarrow \mathbf{q}: j \{\mathbf{m}_i(S_i)\{\mathcal{A}_i\}.G'_i\}_{i \in I} \xrightarrow{\alpha} \langle \mathbb{V}'; \mathbf{p} \rightsquigarrow \mathbf{q}: j \{\mathbf{m}_i(S_i)\{\mathcal{A}_i\}.G''_i\}_{i \in I} \rangle} \text{ [GR-CTX-II]}
\end{array}$$

Figure 7 Top: typing environment semantics. Bottom: timed global type semantics, where $\mathcal{A}_i = \delta_{O,i}, \lambda_{O,i}, \delta_{I,i}, \lambda_{I,i}$.

The label $s:\mathbf{p}!\mathbf{q}:\mathbf{m}$ indicates that \mathbf{p} sends the message \mathbf{m} to \mathbf{q} on session s , while $s:\mathbf{p},\mathbf{q}:\mathbf{m}$ denotes the reception of \mathbf{m} from \mathbf{q} by \mathbf{p} on s . Additionally, the label t ($\in \mathbb{R}_{\geq 0}$) represents a time action modelling the passage of time.

The (highlighted) main modifications in the reduction rules for typing environments, compared to standard rules, concern time. Rule $[\Gamma-\oplus]$ states that an entry can perform an output transition by appending a message at the respective queue within the time specified by the output clock constraint. Dually, rule $[\Gamma-&]$ allows an entry to execute an input transition, consuming a message from the corresponding queue within the specified input clock constraint, provided that the payloads are compatible through subtyping. Note that in both rules, the associated clock valuation of the reduced entry must be updated according to the reset.

Rules $[\Gamma-,x]$ and $[\Gamma-,τ]$ pertain to *untimed* reductions, i.e. $\alpha \neq t$, within a larger environment. Rule $[\Gamma-Ts]$ models time passing on an entry of timed session type by incrementing the associated clock valuation, while rule $[\Gamma-TQ]$ specifies that an entry of queue type is not affected with respect to time progression. Thus, rule $[\Gamma-Tc]$ captures the corresponding time behaviour for a timed-session/queue type entry. Additionally, rule $[\Gamma-,T]$ ensures that time elapses uniformly across compatibly composed environments. Other rules are standard: $[\Gamma-μ]$ is for recursion, and $[\Gamma-STRUCT]$ ensures that reductions are closed under congruence.

The reduction $\Gamma \xrightarrow{s} \Gamma'$ indicates that the typing environment Γ can advance on session s , involving any roles, while $\Gamma \rightarrow \Gamma'$ signifies Γ progressing on any session. This distinction helps in illustrating properties of typed processes discussed in § 4.5.

4.3 Relating Timed Global Types and Typing Environments

One of our main results is establishing an operational relationship between the semantics of timed global types and typing environments, ensuring the *correctness* of processes typed by environments that reflect timed global types. To accomplish this, we begin by assigning LTS semantics to timed global types.

Similar to that of typing environments, we define the LTS semantics for timed global types G over tuples of the form $\langle \mathbb{V}; G \rangle$, where \mathbb{V} is a clock valuation. Additionally, we specify the subject of an action α as its responsible principal: $\text{subject}(s:p!q:m) = \text{subject}(s:p,q:m) = \{p\}$, and $\text{subject}(t) = \emptyset$.

► **Definition 10** (Timed Global Type Reduction). *The timed global type transition $\xrightarrow{\alpha}$ is inductively defined by the rules in Fig. 7 (bottom). We denote $\langle \mathbb{V}; G \rangle \xrightarrow{\alpha} \langle \mathbb{V}'; G' \rangle$ if there exists α such that $\langle \mathbb{V}; G \rangle \xrightarrow{\alpha} \langle \mathbb{V}'; G' \rangle$, $\langle \mathbb{V}; G \rangle \rightarrow$ if there exists $\langle \mathbb{V}'; G' \rangle$ such that $\langle \mathbb{V}; G \rangle \rightarrow \langle \mathbb{V}'; G' \rangle$, and \rightarrow^* as the transitive and reflexive closure of \rightarrow .*

In Fig. 7 (bottom), the (highlighted) changes from the standard global type reduction rules [11] focus on time. Rule [GR-T] accounts for the passage of time by incrementing the clock valuation. Rules [GR-⊕] and [GR-&] model the sending and receiving of messages within specified clock constraints, respectively. Both rules also require the adjustment of the clock valuation using the reset predicate. Rule [GR-μ] handles recursion. Finally, rules [GR-CTX-i] and [GR-CTX-ii] allow reductions of (intermediate) global types causally independent of their prefixes. Note that the execution of any timed global type transition always starts with an initial clock valuation \mathbb{V}^0 , i.e. all clocks in \mathbb{V} are set to 0.

We are now ready to establish a *new* relationship, *association*, between timed global types and typing environments. This association, which is more general than projection (Def. 6) by incorporating subtyping \leqslant (Def. 5), plays a crucial role in formulating the typing rules (§ 4.4) and demonstrating the properties of typed processes (§ 4.5).

► **Definition 11** (Association). *A typing environment Γ is associated with a timed global type $\langle \mathbb{V}; G \rangle$ for a multiparty session s , written $\langle \mathbb{V}; G \rangle \sqsubseteq_s \Gamma$, iff Γ can be split into three (possibly empty) sub-environments $\Gamma = \Gamma_G, \Gamma_\Delta, \Gamma_{\text{end}}$ where:*

1. Γ_G is associated with $\langle \mathbb{V}; G \rangle$ for s , provided as:

- (i) $\text{dom}(\Gamma_G) = \{s[p] \mid p \in \text{roles}(G)\}$;
- (ii) $\forall s[p] \in \text{dom}(\Gamma_G) : \Gamma_G(s[p]) = (\mathbb{V}_p, T_p)$;
- (iii) $\forall p \in \text{roles}(G) : G \upharpoonright p \leqslant T_p$; and
- (iv) $\mathbb{V} = \sqcup_{p \in \text{roles}(G)} \mathbb{V}_p$ (recall that \sqcup is an overriding union).

2. Γ_Δ is associated with G for s , given as follows:

- (i) $\text{dom}(\Gamma_\Delta) = \{s[p] \mid p \in \text{roles}(G)\}$;
- (ii) $\forall s[p] \in \text{dom}(\Gamma_\Delta) : \Gamma_\Delta(s[p]) = \mathcal{M}_p$;
- (iii) if $G = \text{end}$ or $G = \mu t.G'$, then $\forall s[p] \in \text{dom}(\Gamma_\Delta) : \Gamma_\Delta(s[p]) = \emptyset$;
- (iv) if $G = p \rightarrow q : \{m_i(S_i)\{\delta_{O_i}, \lambda_{O_i}, \delta_{I_i}, \lambda_{I_i}\}.G_i\}_{i \in I}$, then
 - (a1) $q \notin \text{receivers}(\Gamma_\Delta(s[p]))$, and
 - (a2) $\forall i \in I : \Gamma_\Delta$ is associated with G_i for s ;
- (v) if $G = p \rightsquigarrow q ; j : \{m_i(S_i)\{\delta_{O_i}, \lambda_{O_i}, \delta_{I_i}, \lambda_{I_i}\}.G_i\}_{i \in I}$, then
 - (b1) $\Gamma_\Delta(s[p]) = q!m_j(S'_j) \cdot \mathcal{M}$ with $S'_j \leqslant S_j$, and
 - (b2) $\Gamma_\Delta[s[p] \mapsto \mathcal{M}]$ is associated with G_j for s .

3. $\forall s[p] \in \text{dom}(\Gamma_{\text{end}}) : \Gamma_{\text{end}}(s[p]) = \emptyset; (\mathbb{V}_p, \text{end})$.

The association $\cdot \sqsubseteq \cdot$ is a binary relation over timed global types $\langle \mathbb{V}; G \rangle$ and typing environments Γ , parameterised by multiparty sessions s . There are three requirements for the association:

- (1) The typing environment Γ must include two entries for each role of the global type G in s : one of timed session type and another of queue type;
- (2) The timed session type entries in Γ reflect $\langle \mathbb{V}; G \rangle$ by ensuring that:
 - a. they align with the projections of G via subtyping, and
 - b. their clock valuations match \mathbb{V} ;
- (3) The queue type entries in Γ correspond to the transmissions en route in G .

Note that Γ_{end} is specifically used to associate typing environments and **end**-types $\langle \mathbb{V}; \text{end} \rangle$, as in this case, both Γ_G and Γ_Δ are empty.

► **Example 12.** Consider the timed global type $\langle \{C_{\text{Sen}} = 0, C_{\text{Sat}} = 0, C_{\text{Ser}} = 0\}; G_{\text{data}} \rangle$, where G_{data} is from Ex. 7, and a typing environment $\Gamma_{\text{data}} = \Gamma_{G_{\text{data}}}, \Gamma_{\Delta_{\text{data}}}$, where:

$$\begin{aligned}\Gamma_{G_{\text{data}}} &= s[\text{Sen}]:(\{C_{\text{Sen}} = 0\}, \text{Sat} \oplus \text{Data}\{6 \leq C_{\text{Sen}} \leq 7, C_{\text{Sen}} := 0\}), \\ s[\text{Sat}] &:(\{C_{\text{Sat}} = 0\}, \text{Sen} \& \left\{ \begin{array}{l} \text{Data}\{6 \leq C_{\text{Sat}} \leq 7, \emptyset\}. \text{Ser} \oplus \text{Data}\{6 \leq C_{\text{Sat}} \leq 7, C_{\text{Sat}} := 0\} \\ \text{fail}\{6 \leq C_{\text{Sat}} \leq 7, \emptyset\}. \text{Ser} \oplus \text{fatal}\{6 \leq C_{\text{Sat}} \leq 7, C_{\text{Sat}} := 0\} \end{array} \right\}), \\ s[\text{Ser}] &:(\{C_{\text{Ser}} = 0\}, \text{Sat} \& \left\{ \begin{array}{l} \text{Data}\{6 \leq C_{\text{Ser}} \leq 7, C_{\text{Ser}} := 0\} \\ \text{fatal}\{6 \leq C_{\text{Ser}} \leq 7, C_{\text{Ser}} := 0\} \end{array} \right\}) \\ \Gamma_{\Delta_{\text{data}}} &= s[\text{Sen}]:\emptyset, s[\text{Sat}]:\emptyset, s[\text{Ser}]:\emptyset\end{aligned}$$

Γ_{data} is associated with $\langle \{C_{\text{Sen}} = 0, C_{\text{Sat}} = 0, C_{\text{Ser}} = 0\}; G_{\text{data}} \rangle$ for s , which can be formally verified by ensuring that Γ_{data} satisfies all conditions outlined in Def. 11.

We establish the operational correspondence between a timed global type and its associated typing environment, our main result for timed multiparty session types, through two theorems: Thm. 13 demonstrates that every possible reduction of a typing environment is mirrored by a corresponding action in reductions of the associated timed global type, while Thm. 14 indicates that the reducibility of a timed global type is equivalent to its associated environment.

► **Theorem 13 (Completeness of Association).** *Given associated timed global type $\langle \mathbb{V}; G \rangle$ and typing environment $\Gamma: \langle \mathbb{V}; G \rangle \sqsubseteq_s \Gamma$. If $\Gamma \xrightarrow{\alpha} \Gamma'$, then there exists $\langle \mathbb{V}'; G' \rangle$ such that $\langle \mathbb{V}; G \rangle \xrightarrow{\alpha} \langle \mathbb{V}'; G' \rangle$ and $\langle \mathbb{V}'; G' \rangle \sqsubseteq_s \Gamma'$.*

► **Theorem 14 (Soundness of Association).** *Given associated timed global type $\langle \mathbb{V}; G \rangle$ and typing environment $\Gamma: \langle \mathbb{V}; G \rangle \sqsubseteq_s \Gamma$. If $\langle \mathbb{V}; G \rangle \rightarrow$, then there exists $\alpha', \mathbb{V}', \langle \mathbb{V}''; G'' \rangle, \Gamma',$ and Γ'' , such that $\langle \mathbb{V}'; G \rangle \sqsubseteq_s \Gamma', \langle \mathbb{V}'; G \rangle \xrightarrow{\alpha'} \langle \mathbb{V}''; G'' \rangle, \Gamma' \xrightarrow{\alpha'} \Gamma'',$ and $\langle \mathbb{V}''; G'' \rangle \sqsubseteq_s \Gamma''$.*

► **Remark 15.** We formulate a soundness theorem that does not mirror the completeness theorem, differing from prior work such as [11]. This choice stems from our reliance on subtyping (Def. 5), notably $[\text{SUB-}\oplus]$. In our framework, a timed local type in the typing environment might offer fewer selection branches compared to the corresponding projected timed local type. Consequently, certain sending actions with their associated clock valuations may remain uninhabited within the timed global type. Consider, e.g. a timed global type:

$$\langle \mathbb{V}_r; G_r \rangle = \langle \{C_p = 3, C_q = 3\}; p \rightarrow q: \left\{ \begin{array}{l} m_1\{0 \leq C_p \leq 1, \emptyset, 1 \leq C_q \leq 2, \emptyset\}. \text{end} \\ m_2\{2 \leq C_p \leq 4, \emptyset, 5 \leq C_q \leq 6, \emptyset\}. \text{end} \end{array} \right\} \rangle$$

An associated typing environment Γ_r may have:

$$\Gamma_r(s[p]) = (\{C_p = 3\}, q \oplus m_1\{0 \leq C_p \leq 1, \emptyset\}. \text{end}); \emptyset \geq (\{C_p = 3\}, q \oplus \left\{ \begin{array}{l} m_1\{0 \leq C_p \leq 1, \emptyset\}. \text{end} \\ m_2\{2 \leq C_p \leq 4, \emptyset\}. \text{end} \end{array} \right\}); \emptyset$$

While the timed global type $\langle \mathbb{V}_r; G_r \rangle$ might transition through $s:p!q:m_2$, the associated environment Γ_r cannot. Nevertheless, our soundness theorem *adequately* guarantees communication safety (communication matches) via association.

$$\begin{array}{c}
 \frac{\Theta(X) = (\mathbb{V}_1, T_1), \dots, (\mathbb{V}_n, T_n)}{\Theta \vdash X:(\mathbb{V}_1, T_1), \dots, (\mathbb{V}_n, T_n)} \text{ [T-X]} \quad \frac{\forall i \in 1..n \quad c_i:(\mathbb{V}_i, T_i) \vdash c_i:(\mathbb{V}'_i, \mathbf{end})}{\text{end}(c_1:(\mathbb{V}_1, T_1), \dots, c_n:(\mathbb{V}_n, T_n))} \text{ [T-end]} \\
 \frac{(\mathbb{V}, T) \leqslant (\mathbb{V}', T')} {\text{c:}(\mathbb{V}, T) \vdash \text{c:}(\mathbb{V}', T')} \text{ [T-SUB]} \quad \frac{\Theta, X:(\mathbb{V}_1, T_1), \dots, (\mathbb{V}_n, T_n) \cdot x_1:(\mathbb{V}_1, T_1), \dots, x_n:(\mathbb{V}_n, T_n) \vdash P}{\Theta, X:(\mathbb{V}_1, T_1), \dots, (\mathbb{V}_n, T_n) \cdot \Gamma \vdash Q} \text{ [T-def]} \\
 \frac{\text{end}(\Gamma)}{\Theta \cdot \Gamma \vdash \mathbf{0}} \text{ [T-0]} \quad \frac{\Theta \vdash X:(\mathbb{V}_1, T_1), \dots, (\mathbb{V}_n, T_n) \quad \text{end}(\Gamma_0) \quad \forall i \in 1..n \quad \Gamma_i \vdash c_i:(\mathbb{V}_i, T_i)}{\Theta \cdot \Gamma_0, \Gamma_1, \dots, \Gamma_n \vdash X \langle c_1, \dots, c_n \rangle} \text{ [T-X-CALL]} \\
 \frac{\forall t \text{ s.t. } \models \delta[t/C] : \Theta \cdot \Gamma \vdash \mathbf{delay}(t) . P}{\Theta \cdot \Gamma \vdash \mathbf{delay}(\delta) . P} \text{ [T-}\delta\text{]} \quad \frac{\Theta \cdot \Gamma + t \vdash P}{\Theta \cdot \Gamma \vdash \mathbf{delay}(t) . P} \text{ [T-t]} \\
 \frac{\forall i \in I \quad \forall t : t \leq n \implies \mathbb{V} + t \models \delta_i \quad \Gamma_1 \vdash c:(\mathbb{V}, \mathbf{q} \& \{ \mathbf{m}_i(S_i) \{ \delta_i, \lambda_i \} . T_i \}_{i \in I}) \quad \forall i \in I : S_i = (\delta'_i, T'_i) \quad \mathbb{V}'_i \models \delta'_i \quad \forall i \in I \quad \forall t \leq n : \Theta \cdot \Gamma + t, y_i:(\mathbb{V}'_i, T'_i), c:(\mathbb{V} + t[\lambda_i \mapsto 0], T_i) \vdash P_i}{\Theta \cdot \Gamma, \Gamma_1 \vdash c^n[\mathbf{q}] \sum_{i \in I} \mathbf{m}_i(y_i).P_i} \text{ [T-&]} \\
 \frac{\forall t : t \leq n \implies \mathbb{V} + t \models \delta \quad \Gamma_1 \vdash c:(\mathbb{V}, \mathbf{q} \oplus \{ \mathbf{m}(S) \{ \delta, \lambda \} . T \}) \quad S = (\delta', T') \quad \Gamma_2 \vdash d:(\mathbb{V}', T') \quad \mathbb{V}' \models \delta' \quad \forall t \leq n : \Theta \cdot \Gamma + t, c:(\mathbb{V} + t[\lambda \mapsto 0], T) \vdash P}{\Theta \cdot \Gamma, \Gamma_1, \Gamma_2 \vdash c^n[\mathbf{q}] \oplus \mathbf{m}(d).P} \text{ [T-}\oplus\text{]} \\
 \frac{\Theta \cdot \Gamma_1 \vdash P_1 \quad \Theta \cdot \Gamma_2 \vdash P_2}{\Theta \cdot \Gamma_1, \Gamma_2 \vdash P_1 | P_2} \text{ [T-|]} \quad \frac{\text{end}(\Gamma) \quad n \geq 0}{\Theta \cdot \Gamma, s[\mathbf{p}_1]:\tau_1, \dots, s[\mathbf{p}_n]:\tau_n \vdash s \notin} \text{ [T-KILL]} \\
 \frac{\Theta \cdot \Gamma \vdash P \quad \text{subjP}(P) = \{c\} \quad \Theta \cdot \Gamma \vdash Q}{\Theta \cdot \Gamma \vdash \mathbf{try} P \mathbf{catch} Q} \text{ [T-TRY]} \quad \frac{\Theta \cdot \Gamma \vdash Q}{\Theta \cdot \Gamma, s[\mathbf{p}]:\tau \mathbf{cancel}(s[\mathbf{p}]).Q} \text{ [T-CANCEL]} \\
 \frac{\Theta \cdot \Gamma \vdash \mathbf{timeout}[P]}{\Theta \cdot \Gamma \vdash \mathbf{timeout}[P]} \text{ [T-FAILED]} \quad \frac{\langle \mathbb{V}; G \rangle \sqsubseteq_s \Gamma' \quad s \notin \Gamma \quad \Theta \cdot \Gamma, \Gamma' \vdash P}{\Theta \cdot \Gamma \vdash (\nu s:\Gamma') P} \text{ [T-}\nu\text{-}G\text{]} \\
 \frac{\text{end}(\Gamma)}{\Theta \cdot \Gamma, s[\mathbf{p}]:\emptyset \vdash s[\mathbf{p}] \blacktriangleright \epsilon} \text{ [T-}\epsilon\text{]} \quad \frac{\Theta \cdot \Gamma \vdash s[\mathbf{p}] \blacktriangleright \sigma \quad S = (\delta, T) \quad \mathbb{V} \models \delta \quad \Gamma' \vdash s'[\mathbf{r}]:(\mathbb{V}, T)}{\Theta \cdot \Gamma[s[\mathbf{p}] \mapsto \mathbf{q}! \mathbf{m}(S) \cdot \Gamma(s[\mathbf{p}])], \Gamma' \vdash s[\mathbf{p}] \blacktriangleright \mathbf{q}! \mathbf{m}(s'[\mathbf{r}]) \cdot \sigma} \text{ [T-}\sigma\text{]}
 \end{array}$$

Figure 8 ATMP typing rules.

4.4 Affine Timed Multiparty Session Typing System

We now present a typing system for ATMP, which relies on *typing judgments* of the form:

$$\Theta \cdot \Gamma \vdash P \quad (\text{with } \Theta \text{ omitted when empty})$$

This judgement indicates that the process P adheres to the usage of its variables and channels as specified in Γ (Def. 8), guided by the process types in Θ (Def. 8). Our typing system is defined inductively by the typing rules shown in Fig. 8, with channels annotated for convenience, especially those bound by process definitions and restrictions.

The innovations (highlighted) in Fig. 8 primarily focus on typing processes with time, timeout failures, message queues, and using association (Def. 11) to enforce session restrictions. **Standard** from [37]: Rule [T-X] retrieves process variables. Rule [T-SUB] applies subtyping within a singleton typing environment $c:(\mathbb{V}, T)$. Rule [T-end] introduces a predicate $\text{end}(\cdot)$ for typing environments, signifying the termination of all endpoints. This predicate is used in [T-0] to type an inactive process $\mathbf{0}$. Rules [T-def] and [T-X-CALL] deal with recursive processes declarations and calls, respectively. Rule [T-|] partitions the typing environment into two, each dedicated to typing one sub-process.

Session Restriction: Rule [T- ν -G] depends on a typing environment associated with a timed global type in a given session s to validate session restrictions.

Delay: Rule [T- δ] ensures the typedness of time-consuming delay $\mathbf{delay}(\delta) . P$ by checking every deterministic delay $\mathbf{delay}(t) . P$ with t as a possible solution to δ . Rule [T-t] types a deterministic delay $\mathbf{delay}(t) . P$ by adjusting the clock valuations in the environment used to type P . Here, $\Gamma + t$ denotes the typing environment obtained from Γ by increasing the associated clock valuation in each entry by t .

Timed Branching and Selection: Rules [T-&] and [T-⊕] are for timed branching and selection, respectively. We elaborate on [T-&], as [T-⊕] is its dual. The first premise in [T-&] specifies a time interval $[\mathbb{V}, \mathbb{V} + n]$ within which the message must be received, in accordance with each δ_i . The last premise requires that each continuation process be well-typed against the continuation of the type in all possible typing environments where the time falls between $[\mathbb{V}, \mathbb{V} + n]$. Here, the clock valuation \mathbb{V} is reset based on each λ_i . The remaining premises stipulate that the clock valuation \mathbb{V}'_i of each delegated receiving session must satisfy δ'_i , and that c is typed.

Try-Catch, Cancellation, and Kill: Rules [T-TRY], [T-CANCEL], and [T-KILL] pertain to try-catch, cancellation, and kill processes, respectively, analogous to the corresponding rules in [28]. [T-CANCEL] is responsible for generating a kill process at its declared session. [T-KILL] types a kill process arising during reductions: it involves broadcasting the cancellation of $s[p]$ to all processes that belong to s . [T-TRY] handles a **try-catch** process **try** P **catch** Q by ensuring that the **try** process P and the **catch** process Q maintain consistent session typing. Additionally, P cannot be a queue or parallel composition, as indicated by $\text{subjP}(P) = \{c\}$.

Timeout Failure: Rule [T-FAILED] indicates that a process raising timeout failure can be typed by *any* typing environment.

Queue: Rules [T-ε] and [T-σ] concern the typing of queues. [T-ε] types an empty queue under an ended typing environment, while [T-σ] types a non-empty queue by inserting a message type into Γ . This insertion may either prepend the message to an existing queue type in Γ or add a queue-typed entry to Γ if not present.

► **Example 16.** Take the typing environment Γ_{data} from Ex. 12, along with the processes Q_{Sen} , Q_{Sat} , Q_{Ser} from Ex. 4. Verifying the typing of $Q_{\text{Sen}} | Q_{\text{Sat}} | Q_{\text{Ser}}$ by Γ_{data} is easy. Moreover, since Γ_{data} is associated with a timed global type $\langle \{C_{\text{Sen}} = 0, C_{\text{Sat}} = 0, C_{\text{Ser}} = 0\}; G_{\text{data}} \rangle$ for session s (as demonstrated in Ex. 12), i.e. $\langle \{C_{\text{Sen}} = 0, C_{\text{Sat}} = 0, C_{\text{Ser}} = 0\}; G_{\text{data}} \rangle \sqsubseteq_s \Gamma_{\text{data}}$, following [T-ν-G], $Q_{\text{Sen}} | Q_{\text{Sat}} | Q_{\text{Ser}}$ is closed under Γ_{data} , i.e. $\vdash (\nu s : \Gamma_{\text{data}}) Q_{\text{Sen}} | Q_{\text{Sat}} | Q_{\text{Ser}}$.

4.5 Typed Process Properties

We demonstrate that processes typed by the ATMP typing system exhibit the desirable properties: *subject reduction* (Thm. 17), *session fidelity* (Thm. 21), and *deadlock-freedom* (Thm. 24).

Subject Reduction. Subject reduction ensures the preservation of well-typedness of processes during reductions. Specifically, it states that if a well-typed process P reduces to P' , this reduction is reflected in the typing environment Γ used to type P . Notably, in our subject reduction theorem, P is constructed from a timed global type, i.e. typed by an environment associated with a timed global type, and this construction approach persists as an invariant property throughout reductions. Furthermore, the theorem does not require P to contain only a single session; instead, it includes all restricted sessions in P , ensuring that reductions on these sessions uphold their respective restrictions. This enforcement is facilitated by rule [T-ν-G] in Fig. 8.

► **Theorem 17** (Subject Reduction). *Assume $\Theta \cdot \Gamma \vdash P$ where $\forall s \in \Gamma : \exists \langle \mathbb{V}; G \rangle : \langle \mathbb{V}; G \rangle \sqsubseteq_s \Gamma_s$. If $P \rightarrow P'$, then $\exists \Gamma'$ such that $\Gamma \xrightarrow{*} \Gamma'$, $\Theta \cdot \Gamma' \vdash P'$, and $\forall s \in \Gamma' : \exists \langle \mathbb{V}'; G' \rangle : \langle \mathbb{V}'; G' \rangle \sqsubseteq_s \Gamma'_s$.*

► **Corollary 18** (Type Safety). *Assume $\emptyset \cdot \emptyset \vdash P$. If $P \xrightarrow{*} P'$, then P' has no communication error.*

► **Example 19.** Take the typed process $Q_{\text{Sen}} | Q_{\text{Sat}} | Q_{\text{Ser}}$ and the typing environment Γ_{data} from Exs. 4, 12, and 16. After a reduction using [R-DET], $Q_{\text{Sen}} | Q_{\text{Sat}} | Q_{\text{Ser}}$ transitions to $\text{delay}(6.5) \cdot Q'_{\text{Sen}} | s[\text{Sen}] \blacktriangleright \epsilon | \text{delay}(6) \cdot Q'_{\text{Sat}} | s[\text{Sat}] \blacktriangleright \epsilon | \text{delay}(6) \cdot Q'_{\text{Ser}} | s[\text{Ser}] \blacktriangleright \epsilon = Q_2$, which

remains typable by Γ_{data} ($\Gamma_{\text{data}} \rightarrow^* \Gamma_{\text{data}}$). Then, applying [R-TIME], Q_2 evolves to $\Psi_{6.5}(Q_2)$, typed as $\Gamma_{\text{data}} + 6.5$, derived from $\Gamma_{\text{data}} \xrightarrow{6.5} \Gamma_{\text{data}} + 6.5$. Further reduction through [R-FAIL] leads $\Psi_{6.5}(Q_2)$ to $Q'_{\text{Sen}} | s[\text{Sen}] \blacktriangleright \epsilon | s[\text{Sat}] \blacktriangleright \epsilon | \Psi_{0.5}(Q'_{\text{Ser}}) | s[\text{Ser}] \blacktriangleright \epsilon = Q_3$, typable by $\Gamma_{\text{data}} + 6.5$. Later, via [C-CAT], Q_3 reduces to $\text{cancel}(s[\text{Sen}]) | s[\text{Sen}] \blacktriangleright \epsilon | s[\text{Sat}] \blacktriangleright \epsilon | \Psi_{0.5}(Q'_{\text{Ser}}) | s[\text{Ser}] \blacktriangleright \epsilon$, which can be typed by Γ''_{data} , obtained from $\Gamma_{\text{data}} + 6.5 \xrightarrow{s:\text{Sen}!\text{Sat}:Data} . \xrightarrow{s:\text{Sat},\text{Sen}:Data} \Gamma''_{\text{data}}$.

Session Fidelity. Session fidelity states the converse implication of subject reduction: if a process P is typed by Γ and Γ can reduce, then P can simulate at least one of the reductions performed by Γ – although not necessarily all such reductions, as Γ over-approximates the behavior of P . Consequently, we can infer P 's behaviour from that of Γ . However, this result does not hold for certain well-typed processes, such as those that get trapped in recursion loops like $\text{def } X(\dots) = X \text{ in } X$, or deadlock due to interactions across multiple sessions [8]. To address this, similarly to [37] and most session type works, we establish session fidelity specifically for processes featuring guarded recursion and implementing a single multiparty session as a parallel composition of one sub-process per role. The formalisation of session fidelity is provided in Thm. 21, building upon the concepts introduced in Def. 20.

► **Definition 20** (From [37]). Assume $\emptyset \cdot \Gamma \vdash P$. We say that P :

1. has guarded definitions if and only if in each process definition in P of the form $\text{def } X(x_1:(\mathbb{V}_1, T_1), \dots, x_n:(\mathbb{V}_n, T_n)) = Q \text{ in } P'$, for all $i \in 1 \dots n$, $T_i \not\leq \text{end}$ implies that a call $Y(\dots, x_i, \dots)$ can only occur in Q as a subterm of $x_i^n[\mathbf{q}] \sum_{j \in \mathbf{m}} (y_j).P_j$ or $x_i^n[\mathbf{q}] \oplus \mathbf{m}\langle d \rangle.P''$ (i.e. after using x_i for input or output);
2. only plays role p in s , by Γ , if and only if
 - (i) P has guarded definitions;
 - (ii) $\text{fv}(P) = \emptyset$;
 - (iii) $\Gamma = \Gamma_0, s[p]:\tau$ with $\tau \not\leq (\mathbb{V}, \text{end})$ and $\text{end}(\Gamma_0)$;
 - (iv) in all subterms $(\nu s':\Gamma') P'$ of P , we have $\Gamma' \leq s'[p']:\emptyset; (\mathbb{V}', \text{end})$ or $\Gamma' \leq s'[p']:(\mathbb{V}', \text{end})$ (for some p', \mathbb{V}').

We say “ P only plays role p in s ” if and only if $\exists \Gamma : \emptyset \cdot \Gamma \vdash P$, and item 2 holds.

In Def. 20, item 1 describes guarded recursion for processes, while item 2 specifies a process limited to playing exactly *one* role within *one* session, preventing an ensemble of such processes from deadlocking by waiting for each other on multiple sessions.

We proceed to present our session fidelity result, taking kill processes into account. We denote $Q \not\leq$ to indicate that Q consists only of a parallel composition of kill processes. Similar to subject reduction (Thm. 17), our session fidelity relies on a typing environment associated with a timed global type for a specific session s to type the process, ensuring the fulfilment of single-session requirements (Def. 20) and maintaining invariance during reductions.

► **Theorem 21** (Session Fidelity). Assume $\emptyset \cdot \Gamma \vdash P$, with $\langle \mathbb{V}; G \rangle \sqsubseteq_s \Gamma$, $P \equiv \Pi_{p \in I} P_p \mid Q \not\leq$, and $\Gamma = \bigcup_{p \in I} \Gamma_p \cup \Gamma_0$, such that $\emptyset \cdot \Gamma_0 \vdash Q \not\leq$, and for each P_p :

- (1) $\emptyset \cdot \Gamma_p \vdash P_p$, and
 - (2) either $P_p \equiv \mathbf{0}$, or P_p only plays role p in s , by Γ_p .
- Then, $\Gamma \rightarrow_s$ implies $\exists \Gamma', \langle \mathbb{V}'; G' \rangle, P'$ such that $\Gamma \rightarrow_s \Gamma'$, $P \rightarrow^* P'$, and $\emptyset \cdot \Gamma' \vdash P'$, with $\langle \mathbb{V}'; G' \rangle \sqsubseteq_s \Gamma'$, $P' \equiv \Pi_{p \in I} P'_p \mid Q' \not\leq$, and $\Gamma' = \bigcup_{p \in I} \Gamma'_p \cup \Gamma'_0$ such that $\emptyset \cdot \Gamma'_0 \vdash Q' \not\leq$, and for each P'_p :
- (1) $\emptyset \cdot \Gamma'_p \vdash P'_p$, and
 - (2) either $P'_p \equiv \mathbf{0}$, or P'_p only plays role p in s , by Γ'_p .

► **Example 22.** Consider the processes Q_{Sen} , Q_{Sat} , Q_{Ser} from Ex. 4, the process Q_3 from Ex. 19, and the typing environment Γ_{data} from Ex. 12. Q_{Sen} , Q_{Sat} , and Q_{Ser} only play roles Sen , Sat , and Ser , respectively, in s , which can be easily verified. As demonstrated

in Ex. 19, Q_3 is typed by $\Gamma_{\text{data}} + 6.5$, satisfying all prerequisites specified in Thm. 21. Consequently, given $\Gamma_{\text{data}} + 6.5 \xrightarrow{s:\text{Sen!Sat:Data}} \Gamma'_{\text{data}}$, there exists Q_4 , resulting from $Q_3 \rightarrow Q_4$ via [R-OUT], such that Γ'_{data} can type Q_4 , with Γ'_{data} and Q_4 fulfilling the single session requirements of session fidelity.

Deadlock-Freedom. Deadlock-freedom ensures that a process can always either progress via reduction or terminate properly. In our system, where time can be infinitely reduced and session killings may occur during reductions, deadlock-freedom implies that if a process cannot undergo any further instantaneous (communication) reductions, and if any subsequent time reduction leaves it unchanged, then it contains only inactive or kill sub-processes. This desirable runtime property is guaranteed by processes constructed from timed global types. We formalise the property in Def. 23, and conclude, in Thm. 24, that a typed ensemble of processes interacting on a single session, restricted by a typing environment Γ associated with a timed global type $\langle \mathbb{V}^0; G \rangle$, is deadlock-free.

► **Definition 23** (Deadlock-Free Process). P is deadlock-free if and only if $P \rightarrow^* P' \not\rightarrow$ and $\forall t \geq 0 : \Psi_t(P') = P'$ (recall that $\Psi_t(\cdot)$ is a time-passing function defined in Fig. 4) implies $P' \equiv \mathbf{0} \mid \prod_{i \in I} s_i \not\downarrow$.

► **Theorem 24** (Deadlock-Freedom). Assume $\emptyset \cdot \emptyset \vdash P$, where $P \equiv (\nu s : \Gamma) \prod_{p \in \text{roles}(G)} P_p$, $\langle \mathbb{V}^0; G \rangle \sqsubseteq_s \Gamma$, and each P_p is either $\mathbf{0}$ (up to \equiv), or only plays p in s . Then, P is deadlock-free.

► **Example 25.** Given the processes Q_{Sen} , Q_{Sat} , and Q_{Ser} from Ex. 4, along with the typing environment Γ_{data} from Ex. 12, $(\nu s : \Gamma_{\text{data}}) Q_{\text{Sen}} \mid Q_{\text{Sat}} \mid Q_{\text{Ser}}$ is deadlock-free.

5 Design and Implementation of MultiCrusty^T

In this section, we present our toolchain, MultiCrusty^T, a RUST implementation of ATMP. MultiCrusty^T is designed with two main goals: correctly cascading failure notifications, and effectively handling time constraints. To achieve the first goal, we use RUST’s native $\text{?}-\text{operator}$ along with optional types, inspired by [28]. For the second objective, we begin by discussing the key challenges encountered during implementation.

Challenge 1: Representation of Time Constraints. To handle asynchronous timed communications using ATMP, we define a time window (δ in ATMP) and a corresponding behaviour for each operation. Addressing this constraint involves two subtasks: creating and using clocks in RUST, and representing all clock constraints as shown in § 3. RUST allows the creation of virtual clocks that rely on the CPU’s clock and provide nanosecond-level accuracy. Additionally, it is crucial to ensure that different behaviours can involve blocking or non-blocking communications, pre- or post-specific time tags, or adherence to specified time bounds.

Challenge 2: Enforcement of Time Constraints. To effectively enforce time windows, implementing reliable and accurate clocks and using them correctly is imperative. This requires addressing all cases related to time constraints properly: clocks may be considered unreliable if they skip ticks, do not strictly increase, or if the API for clock comparison does not yield results quickly enough. Enforcing time constraints in MultiCrusty^T involves using two libraries: the `crossbeam_channel` RUST library [9] for *asynchronous* messaging, and the RUST standard library `time` [39] for handling and comparing virtual clocks.

5.1 Time Bounds in MultiCrusty^T

Implementing Time Bounds. To demonstrate the integration of time bounds in MultiCrusty^T, we consider the final interaction between **Sen** and **Sat** in Fig. 1b, specifically from **Sat**'s perspective: **Sat** sends a **Close** message between time units 5 and 6 (both inclusive), following clock $C_{\text{Sat}2}$, which is not reset afterward.

In MultiCrusty^T, we define the **Send** type for message transmission, incorporating various parameters to specify requirements as `Send<[parameter1], [parameter2], ...>`. Assuming the (payload) type **Close** is defined, sending it using the **Send** type initiates with `Send<Close, ...>`. If $C_{\text{Sat}2}$ is denoted as '**b**', the clock '**b**' is employed for time constraints, expressed as `Send<Close, 'b', ...>`. Time bounds parameters in the **Send** type follow the clock declaration. In this case, both bounds are integers within the time window, resulting in the **Send** type being parameterised as `Send<Close, 'b', 0, true, 1, false, ...>`. Notably, bounds are integers due to the limitations of RUST's generic type parameters. To ensure that the clock '**b**' is not reset after triggering the **send** operation, we represent this with a whitespace char value in the **Send** type: `Send<Close, 'b', 0, true, 1, false, ' ', ...>`. The last parameter, known as the *continuation*, specifies the operation following the sending of the integer. In this case, closing the connection is achieved with an **End** type. The complete sending operation is denoted as `Send<Close, 'b', 0, true, 1, false, ' ', End>`.

Similarly, the **Recv** type is instantiated as `Recv<Close, 'b', 0, true, 1, false, ' ', End>`. The inherent mirroring of **Send** and **Recv** reflects their dual compatibility. Figs. 2a and 2b provide an analysis of the functioning of **Send** and **Recv**, detailing their parameters and features. Generic type parameters preceded by **const** within **Send** and **Recv** types also serve as values, representing general type categories supported by RUST. This type-value duality facilitates easy verification during compilation, ensuring compatibility between communicating parties.

Enforcing Time Bounds. It is crucial to rely on dependable clocks and APIs to enforce time constraints. RUST's standard library provides the time module [39], enabling developers to manage clocks and measure durations between events. This library, utilising the OS API, offers two clock types: **Instant** (monotonic but non-steady) and **SystemTime** (steady but non-monotonic). In MultiCrusty^T, the **Instant** type serves for both correctly prioritising event order and implementing virtual clocks. Virtual clocks are maintained through a dictionary (**HashMap** in RUST). Table 1 details the primitives provided by MultiCrusty^T for sending and receiving payloads, implementing branching, or closing connections. All primitives, except for **close**, require a specific **HashMap** of clocks to enforce time constraints.

Verifying Time Bounds. Our **send** and **recv** primitives use a series of conditions to ensure the integrity of a time window. The verification process adopts a *divide-and-conquer* strategy, validating the left-hand side time constraint for each clock before assessing the right-hand side constraint. The corresponding operation, whether sending or receiving a payload, is executed only after satisfying these conditions. This approach guarantees the effective enforcement of time constraints without requiring complex solver mechanisms.

5.2 Remote Data Implementation

Implementation of Server. Fig. 9 explores our MultiCrusty^T implementation of **Ser** in the remote data protocol (Fig. 1b). Specifically, the left side of Fig. 9 delves into the **MeshedChannels** type, representing the behaviour of **Ser** in the first branch and encapsulating various elements. In MultiCrusty^T, the **MeshedChannels** type incorporates $n + 1$ parameters,

■ **Table 1** Primitives available in MultiCrusty^T.

<code>let s = s.send(p, clocks)?;</code>	If allowed by the time constraint compared to the given clock in <code>clocks</code> , sends a payload <code>p</code> on a channel <code>s</code> and assigns the continuation of the session (a new meshed channel) to a new variable <code>s</code> .
<code>let (p, s) = s.recv(clocks)?;</code>	If allowed by the time constraint compared to the given clock in <code>clocks</code> , receives a payload <code>p</code> on channel <code>s</code> and assigns the continuation of the session to a new variable <code>s</code> .
<code>s.close()</code>	Closes the channel <code>s</code> and returns a unit type.
<code>offer!(s, clocks, { enum_i :: variant_k(e) => {...} }_{k ∈ K} }</code>	If allowed by the time constraint compared to the given clock in <code>clocks</code> , role <code>i</code> receives a choice as a message label on channel <code>s</code> , and, depending on the label value which should match one of the variants <code>variant_k</code> of <code>enum_i :: ...</code> , runs the related block of code.
<code>choose_X!(s, clocks, { enum_i :: variant_k(e) }_{i ∈ I})</code>	For role <code>X</code> , if allowed by the time constraint compared to the given clock in <code>clocks</code> , sends the chosen label, corresponding to <code>variant_k</code> to all other roles.

```

1  type EndpointSerData = MeshedChannels<
2    Send<GetData, 'a', 5, true, 5, true, ' ', ,
3      Recv<Data, 'a', 6, true, 7, true, 'a', End
4      >>,
5    End,
6    RoleSat<RoleSat<RoleBroadcast>>,
7    fn endpoint_data_ser(
8      s: EndpointSerData,
9      clocks: &mut HashMap<char, Instant>,
10     ) -> Result<(), Error> { [...] }
11    let s = s.send(GetData {}, clocks)?;
12    let (_data, s) = s.recv(clocks)?; [...]
6    NameSer>;

```

■ **Figure 9** Types (left) and primitives (right) for `Ser`.

where n is the count of roles in the protocol. These parameters include the role's name, $n - 1$ binary channels for interacting with other roles, and a stack dictating the sequence of binary channel usage. All types relevant to `Ser` are depicted in Fig. 9 (left).

The alias `EndpointSerData`, as indicated in Line 1, represents the `MeshedChannels` type. Binary types, defined in Lines 2–4, facilitate communication between `Ser`, `Sat`, and `Sen`. When initiating communication with `Sat`, `Ser` sends a `GetData` message in Line 2, receives a `Data` response, and ends communication on this binary channel. These operations use the clock '`a`' and adhere to time windows between 5 and 6 seconds for the first operation and between 6 and 7 seconds for the second. Clock '`a`' is reset only within the second operation. The order of operations is outlined in Line 5, where `Ser` interacts twice with `Sat` using `RoleSat` before initiating a choice with `RoleBroadcast`. Line 6 designates `Ser` as the owner of the `MeshedChannels` type. The behaviour of all roles in each branch can be specified similarly.

The right side of Fig. 9 illustrates the usage of `EndpointSerData` as an input type in the RUST function `endpoint_data_ser`. The function's output type, `Result<(), Error>`, indicates the utilization of affinity in RUST. In Line 11, variable '`s`', of type `EndpointSerData`, attempts to send a contentless message `GetData`. The `send` function can return either a value resembling `EndpointSerData` or an `Error`. If the clock's time does not adhere to the time constraint displayed in Line 2 with respect to the clock '`a`' from the set of clocks `clocks`, an `Error` is raised. Similarly, in Line 12, `Ser` attempts to receive a message using the same set of clocks. Both `send` and `recv` functions verify compliance with time constraints by comparing the relevant clock provided in the type for the time window and resetting the clock if necessary.

Error Handling. The error handling capabilities of MultiCrusty^T cover various potential errors that may arise during protocol implementation and execution. These errors include the misuse of generated types and timeouts, showcasing the flexibility of our implementation

```

1  global protocol RemoteData(role Sen, role Sat, role Ser){
2    rec Loop {
3      choice at Ser {
4        GetData() from Ser to Sat within [5;6] using a and resetting ();
5        GetData() from Sat to Sen within [5;6] using b and resetting ();
6        Data() from Sen to Sat within [6;7] using b and resetting (b);
7        Data() from Sat to Ser within [6;7] using a and resetting (a);
8        continue Loop
9    } or {
10      Close() from Ser to Sat within [5;6] using a and resetting ();
11      Close() from Sat to Sen within [5;6] using b and resetting (); } } }
```

Figure 10 Remote data protocol in νSCR^T .

in verifying communication protocols. For instance, if Lines 11 and 12 in Fig. 9 are swapped, the program will fail to compile because it expects a `send` primitive in Line 11, as indicated by the type of '`s`'. Another compile-time error occurs when a payload with the wrong type is sent. For example, attempting to send a `Data` message instead of a `GetData` in Line 11 will result in a compilation error. `MultiCrustyT` can also identify errors at runtime. If the content of the function `endpoint_data_ser`, spanning in Lines 10–12, is replaced with a single `Ok()`, the code will compile successfully. However, during runtime, the other roles will encounter failures as they consider `Ser` to have failed.

Timeouts are handled dynamically within `MultiCrustyT`. If a time-consuming task with a 10-second delay is introduced between Lines 11 and 12, `Ser` will enter a sleep state for the same duration. Consequently, the `recv` operation in Line 12 will encounter a time constraint violation, resulting in the failure and termination of `Ser`. Furthermore, the absence of clock '`a`' in the set of clocks, where it is required for a specific primitive, will trigger a runtime error, as the evaluation of time constraints depends on the availability of the necessary clocks.

Timed Protocol Specification. To specify timed multiparty protocols, we extend νSCR [42], a multiparty protocol description language, with time features, resulting in νSCR^T . Additional keywords such as `within`, `using`, and `and resetting` are incorporated in νSCR to support the specification of time windows, clocks, and resets, respectively. In Fig. 10, we illustrate the νSCR^T protocol for remote data, showcasing the application of these enhancements. νSCR^T ensures the accuracy of timed multiparty protocols by verifying interactions, validating time constraints, handling clock increments, and performing standard MPST protocol checks.

6 Evaluation: Expressiveness, Case Studies and Benchmarks

We evaluate our toolchain `MultiCrustyT` from two perspectives: *expressivity* and *feasibility*. In terms of expressivity, we implement protocols from the session type literature [20, 33, 13, 24, 21, 36], as well as newly introduced protocols derived from real-world applications [7, 38, 2, 35, 41]. Regarding feasibility, we compare our system to `MultiCrusty` [28], an untimed implementation of affine synchronous MPST, demonstrating that our tool introduces negligible compile-time and runtime overhead in all cases, as expected.

6.1 Performance: `MultiCrustyT` vs. `MultiCrusty`

When comparing `MultiCrustyT` with `MultiCrusty`, we evaluate their performance on two standard benchmark protocols: the *ring* protocol, which involves sequentially passing a message through roles, and the *mesh* protocol, where each participant sends a message to every other. Both protocols underwent 100 iterations within a time window of 0 to 10 seconds. Fig. 11 (top) displays benchmark results for roles ranging from 2 to 8.

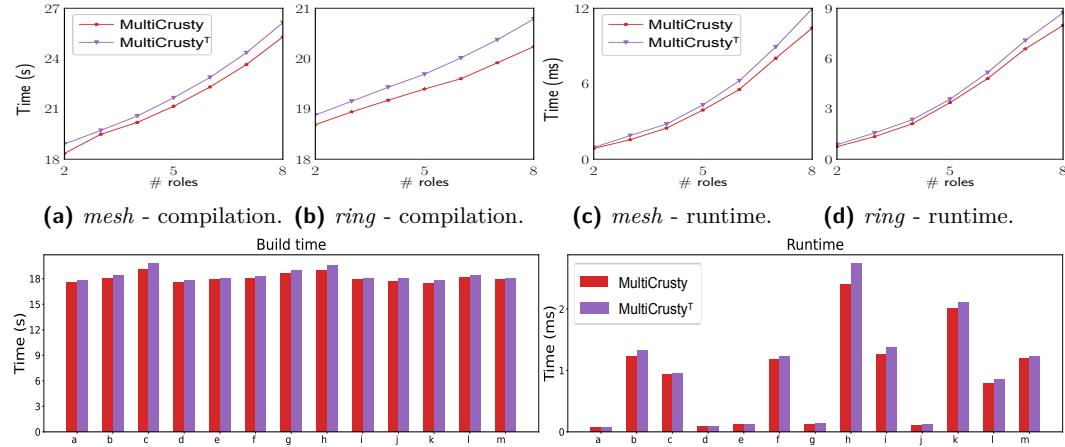


Figure 11 Top: microbenchmark results for mesh and ring protocols. Bottom: benchmark results for Calculator [20] (a), Online wallet [33] (b), SMTP [36] (c), Simple voting [20] (d), Three buyers [24] (e), Travel agency [21] (f), OAuth [33] (g), HTTP [13] (h), Remote data [7] (i), Servo [38] (j), Gravity sensor [2] (k), PineTime heart rate [35] (l), and Proximity based car key [41] (m).

In the *ring* protocol, compile-time benchmarks (Fig. 11b) indicate that *MultiCrusty^T* experiences a marginal slowdown of less than 2% with 2 roles, but achieves approximately 5% faster compilation time with 8 roles. Regarding runtime benchmarks (Fig. 11d), *MultiCrusty* demonstrates a 15% speed advantage with 2 roles, which decreases to 5% with 8 roles. The overhead remains consistent, with a difference of less than 0.5 ms at 6, 7, and 8 roles.

In the *mesh* protocol, where all roles send and receive messages (compile-time benchmarks in Fig. 11a and runtime benchmarks in Fig. 11c), *MultiCrusty^T* compiles slightly slower (less than 1% at 2 roles, 4% at 8 roles) and runs slower as well (less than 1% at 2 roles, 15% at 8 roles). Compile times for *MultiCrusty^T* range from 18.9 s to 26 s, with running times ranging between 0.9 ms and 11.9 ms. The performance gap widens exponentially with the increasing number of enforced time constraints. In summary, as the number of roles increases, *MultiCrusty^T* demonstrates a growing overhead, mainly attributed to the incorporation of additional time constraint checks.

6.2 Expressivity and Feasibility with Case Studies

We implement a variety of protocols to showcase the expressivity, feasibility, and capabilities of *MultiCrusty^T*, conducting benchmarking using both *MultiCrusty^T* and *MultiCrusty*. The `send` and `recv` operations in both libraries are ordered, directed, and involve the same set of participants. Additionally, when implemented with *MultiCrusty^T*, these operations are enriched with time constraints and reset predicates. The benchmark results for the selected case studies, including those from prior research and five additional protocols sourced from industrial use cases [7, 38, 2, 35, 41], are presented in the bottom part of Fig. 11. To ensure a fair comparison between *MultiCrusty^T* (purple) and *MultiCrusty* (red), time constraints are enforced for all examples without introducing any additional sleep or timeouts.

Note that rate-based protocols ((k), (l), (m) in Fig. 11 (bottom)) from real-time systems [2, 35, 41] are implemented in *MultiCrusty^T*, showcasing its expressivity in real-time applications. These implementations feature the establishment of consistent time constraints and a shared clock for operations with identical rates. For example, in the Car Key protocol [41], where the car periodically sends a wake-up message to probe the presence of the key, all interactions

between two wake-up signals must occur within a period of e.g. 100 ms. Consequently, when implementing this protocol with MultiCrusty^T , all time constraints are governed by a single clock ranging from 0 to 100 ms, with the clock resetting at the end of each loop.

The feasibility of our tool, MultiCrusty^T , is demonstrated in Fig. 11 (bottom). The results indicate that MultiCrusty^T incurs minimal compile-time overhead, averaging approximately 1.75%. Moreover, the runtime for each protocol remains within milliseconds, ensuring negligible impact. Notably, in the HTTP protocol, the runtime comparison percentage with MultiCrusty is 87.60%, primarily due to the integration of 126 time constraints within it. The relevant implementation metrics, including multiple participants (MP), branching, recursion (Rec), and time constraints, are illustrated in Table 2.

■ **Table 2** Metrics for protocols implemented in MultiCrusty^T .

Protocol	Generated Types	Implemented Lines of Code	MP	Branching	Rec	Time Constraints
Calculator [20]	52	51	✗	✓	✓	11
Online wallet [33]	142	160	✓	✓	✓	24
SMTP [36]	331	475	✗	✓	✓	98
Simple voting [20]	73	96	✗	✓	✗	16
Three buyers [24]	108	78	✓	✓	✗	22
Travel agency [21]	148	128	✓	✓	✓	30
OAuth [33]	199	89	✓	✓	✗	30
HTTP [13]	648	610	✓	✓	✓	126
Remote data [7]	100	119	✓	✓	✓	16
Servo [38]	74	48	✓	✗	✗	10
Gravity sensor [2]	61	95	✗	✓	✓	9
PineTime heart rate [35]	101	111	✗	✓	✓	17
Proximity based car key [41]	70	134	✗	✓	✓	22

7 Related Work and Conclusion

Time in Session Types. Bocchi et al. [4] propose a timed extension of MPST to model real-time choreographic interactions, while Bocchi et al. [3] extend *binary* session types with time constraints, introducing a subtyping relation and a blocking receive primitive with timeout in their calculus. In contrast to their strategies to avoid time-related failures, as discussed in §1 and 2, ATMP focuses on actively managing failures as they occur, offering a distinct approach to handling timed communication.

Iraci et al. [22] extend *synchronous binary* session types with a periodic recursion primitive to model rate-based processes. To align their design with real-time systems, they encode time into a periodic construct, synchronised with a global clock. With *rate compatibility*, a relation that facilitates communication between processes with different periods by synthesising and verifying a common superperiod type, their approach ensures that well-typed processes remain free from rate errors during any specific period. On the contrary, ATMP integrates time constraints directly into communication through local clocks, resulting in distinct time behaviour. Intriguingly, our method of time encoding can adapt to theirs, while the opposite is not feasible. Consequently, not all the timed protocols in our paper, e.g. Fig. 1b, can be accurately represented in their system. Moreover, due to its *binary* and *synchronous* features, their theory does not directly model and ensure the properties of real distributed systems.

Le Brun et al. [30] develop a theory of multiparty session types that accounts for different failure scenarios, including message losses, delays, reordering, as well as link failures and network partitioning. Unlike ATMP, their approach does not integrate time specifications or address failures specifically related to time. Instead, they use *timeout* as a generic message label (\odot) for failure branches, which triggers the failure detection mechanism. Except for [22], all the mentioned works on session types with time are purely theoretical.

Affinity, Exceptions and Error-Handling in Session Types. Mostrous and Vasconcelos [31] propose affine binary session types with explicit cancellation, which Fowler et al. [14] extend to define Exceptional GV for binary asynchronous communication. Exceptions can be nested and handled over multiple communication actions, and their implementation is an extension of the research language LINKS. Harvey et al. [15] incorporate MPST with explicit connection actions to facilitate multiparty distributed communication, and develop a code generator based on the actor-like research language ENSEMBLE to implement their approach. The work in [31] remains theoretical, and both [31, 14] are limited to binary and linear logic-based session types. Additionally, none of these works considers time specifications or addresses the handling of time-related exceptions in their systems, which are key aspects of our work.

Session Types in Rust. MultiCrusty, extensively compared to MultiCrusty^T, is a RUST implementation based on affine MPST by Lagaillardie et al. [28]. Their approach relies on *synchronous* communication, rendering time and timeout exceptions unnecessary.

Cutner et al. [10] introduce Rumpsteak, a RUST implementation based on the `tokio` RUST library, which uses a different design for asynchronous multiparty communications compared to MultiCrusty^T, relying on the `crossbeam_channel` RUST library. The main goal of [10] is to compare the performance of Rumpsteak, mainly designed to analyse asynchronous message reordering, to existing tools such as the *k-MC* tool developed in [29]. Unlike MultiCrusty^T, Rumpsteak lacks formalisation, or handling of timed communications and failures.

Typestate is a RUST library implemented by Duarte and Ravara [12], focused on helping developers to write safer APIs using typestates and their macros `#[typestate]`, `#[automaton]` and `#[state]`. MultiCrusty^T and Typestate are fundamentally different, with Typestate creating a state machine for checking possible errors in APIs and not handling affine or timed communications. Ferrite, a RUST implementation introduced by Chen et al. [6], is limited to binary session types and forces the use of linear channels. The modelling of Ferrite is based on the shared binary session type calculus SILL_s.

Jespersen et al. [23] and Kokke [25] propose RUST implementations of binary session types for synchronous communication protocols. [22] extends the framework from [23] to encode the *rate compatibility* relation as a RUST trait and check whether two types are rate compatible. Their approach is demonstrated with examples from rate-based systems, including [2, 35, 41]. Motivated by these applications, we formalise and implement the respective timed protocols in MultiCrusty^T, showcasing the expressivity and feasibility of our system in real-time scenarios.

Conclusion and Future Work. To address time constraints and timeout exceptions in asynchronous communication, we propose *affine timed multiparty session types* (ATMP) along with the toolchain MultiCrusty^T, an implementation of ATMP in RUST. Thanks to the incorporation of affinity and failure handling mechanisms, our approach renders impractical conditions such as *wait-freedom* and *urgent receive* obsolete while ensuring communication safety, protocol conformance, and deadlock-freedom, even in the presence of (timeout) failures. Compared to a synchronous toolchain without time, MultiCrusty^T exhibits negligible overhead in various complex examples including those from real-time systems, while enabling the verification of time constraints under asynchronous communication. As future work, we plan to explore automatic recovery from errors and timeouts instead of simply terminating processes, which will involve extending the analysis of communication causality to timed global types and incorporating reversibility mechanisms into our system.

References

- 1 Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994. doi:[10.1016/0304-3975\(94\)90010-8](https://doi.org/10.1016/0304-3975(94)90010-8).
- 2 Android. Motion Sensors, 2009. URL: https://developer.android.com/guide/topics/sensors/sensors_motion.
- 3 Laura Bocchi, Maurizio Murgia, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. Asynchronous timed session types. In Luís Caires, editor, *Programming Languages and Systems*, pages 583–610, Cham, 2019. Springer International Publishing.
- 4 Laura Bocchi, Weizhen Yang, and Nobuko Yoshida. Timed multiparty session types. In Paolo Baldan and Daniele Gorla, editors, *CONCUR 2014 - Concurrency Theory - 25th International Conference, CONCUR 2014, Rome, Italy, September 2-5, 2014. Proceedings*, volume 8704 of *Lecture Notes in Computer Science*, pages 419–434. Springer, 2014. doi:[10.1007/978-3-662-44584-6_29](https://doi.org/10.1007/978-3-662-44584-6_29).
- 5 David Castro, Raymond Hu, SungShik Jongmans, Nicholas Ng, and Nobuko Yoshida. Distributed Programming Using Role-Parametric Session Types in Go: Statically-Typed Endpoint APIs for Dynamically-Instantiated Communication Structures. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. Place: New York, NY, USA Publisher: Association for Computing Machinery. doi:[10.1145/3290342](https://doi.org/10.1145/3290342).
- 6 Ruo Fei Chen, Stephanie Balzer, and Bernardo Toninho. Ferrite: A Judgmental Embedding of Session Types in Rust. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:28, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:[10.4230/LIPIcs.ECOOP.2022.22](https://doi.org/10.4230/LIPIcs.ECOOP.2022.22).
- 7 Yingying Chen, Minghu Zhang, Xin Li, Tao Che, Rui Jin, Jianwen Guo, Wei Yang, Baosheng An, and Xiaowei Nie. Satellite-enabled internet of remote things network transmits field data from the most remote areas of the tibetan plateau. *Sensors*, 22(10):3713, 2022. doi:[10.3390/S22103713](https://doi.org/10.3390/S22103713).
- 8 Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global progress for dynamically interleaved multiparty sessions. *Mathematical Structures in Computer Science*, 26(2):238–302, 2016. doi:[10.1017/S0960129514000188](https://doi.org/10.1017/S0960129514000188).
- 9 The Developers of Crossbeam. Crate: Crossbeam channel, 2022. Last accessed: October 2022. URL: <https://crates.io/crates/crossbeam-channel>.
- 10 Zak Cutner, Nobuko Yoshida, and Martin Vassor. Deadlock-Free Asynchronous Message Reordering in Rust with Multiparty Session Types. In *27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume abs/2112.12693 of *PPoPP '22*, pages 261–246. ACM, 2022. doi:[10.1145/3503221.3508404](https://doi.org/10.1145/3503221.3508404).
- 11 Pierre-Malo Deniéou and Nobuko Yoshida. Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types. In *40th International Colloquium on Automata, Languages and Programming*, volume 7966 of *LNCS*, pages 174–186, Berlin, Heidelberg, 2013. Springer. doi:[10.1007/978-3-642-39212-2_18](https://doi.org/10.1007/978-3-642-39212-2_18).
- 12 José Duarte and António Ravara. Taming stateful computations in rust with typestates. *Journal of Computer Languages*, 72:101154, 2022. doi:[10.1016/j.cola.2022.101154](https://doi.org/10.1016/j.cola.2022.101154).
- 13 Roy Fielding and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing. Technical Report RFC7230, RFC Editor, June 2014. doi:[10.17487/rfc7230](https://doi.org/10.17487/rfc7230).
- 14 Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional Asynchronous Session Types: Session Types Without Tiers. *Proc. ACM Program. Lang.*, 3(POPL):28:1–28:29, January 2019. Place: New York, NY, USA Publisher: ACM. doi:[10.1145/3290341](https://doi.org/10.1145/3290341).
- 15 Paul Harvey, Simon Fowler, Ornella Dardha, and Simon J. Gay. Multiparty Session Types for Safe Runtime Adaptation in an Actor Language. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, volume 194 of *Leibniz International Proceedings in Informatics (LIPIcs)*, page 30, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:[10.4230/LIPIcs.ECOOP.2021.10](https://doi.org/10.4230/LIPIcs.ECOOP.2021.10).

- 16 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. doi:10.1007/BFB0053567.
- 17 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 273–284. ACM, 2008. Full version in [18]. doi:10.1145/1328438.1328472.
- 18 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. *J. ACM*, 63(1), 2016. doi:10.1145/2827695.
- 19 Ping Hou, Nicolas Lagaillardie, and Nobuko Yoshida. Fearless asynchronous communications with timed multiparty session protocols, 2024. arXiv:2406.19541.
- 20 Raymond Hu and Nobuko Yoshida. Hybrid Session Verification Through Endpoint API Generation. In Perdita Stevens and Andrzej Wasowski, editors, *Fundamental Approaches to Software Engineering*, volume 9633, pages 401–418. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. doi:10.1007/978-3-662-49665-7_24.
- 21 Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-Based Distributed Programming in Java. In Jan Vitek, editor, *ECOOP'08*, volume 5142 of *LNCS*, pages 516–541, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. doi:10.1007/978-3-540-70592-5_22.
- 22 Grant Iraci, Cheng-En Chuang, Raymond Hu, and Lukasz Ziarek. Validating iot devices with rate-based session types. *Proc. ACM Program. Lang.*, 7(OOPSLA2):1589–1617, 2023. doi:10.1145/3622854.
- 23 Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. Session Types for Rust. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, WGP 2015, pages 13–22, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2808098.2808100.
- 24 Limin Jia, Hannah Gommerstadt, and Frank Pfenning. Monitors and Blame Assignment for Higher-Order Session Types. *SIGPLAN Not.*, 51(1):582–594, January 2016. doi:10.1145/2914770.2837662.
- 25 Wen Kokke. Rusty Variation: Deadlock-free Sessions with Failure in Rust. *Electronic Proceedings in Theoretical Computer Science*, 304:48–60, September 2019. doi:10.4204/eptcs.304.4.
- 26 Dimitrios Kouzapas, Ornella Dardha, Roly Perera, and Simon J. Gay. Typechecking Protocols with Mungo and stmungo. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*, PPDP '16, pages 146–159, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2967973.2968595.
- 27 Pavel Krcál and Wang Yi. Communicating timed automata: The more synchronous, the more difficult to verify. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 249–262. Springer, 2006. doi:10.1007/11817963_24.
- 28 Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. Stay Safe Under Panic: Affine Rust Programming with Multiparty Session Types. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:29, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2022.4.
- 29 Julien Lange and Nobuko Yoshida. Verifying Asynchronous Interactions via Communicating Session Automata. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019*, volume 11561 of *Lecture Notes in Computer Science*, pages 97–117, Cham, 2019. Springer. doi:10.1007/978-3-030-25540-4_6.

19:30 Fearless Asynchronous Communications with Timed Multiparty Session Protocols

- 30 Matthew Alan Le Brun and Ornella Dardha. MAG π : Types for Failure-Prone Communication. In Thomas Wies, editor, *Programming Languages and Systems*, pages 363–391, Cham, 2023. Springer Nature Switzerland. doi:10.1007/978-3-031-30044-8_14.
- 31 Dimitris Mostrous and Vasco T. Vasconcelos. Affine Sessions. *Logical Methods in Computer Science* ; Volume 14, 8459:Issue 4 ; 18605974, 2018. Medium: PDF Publisher: Episciences.org. doi:10.23638/LMCS-14(4:14)2018.
- 32 Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. Timed runtime monitoring for multiparty conversations. *Formal Aspects of Computing*, 29(5):877–910, 2017.
- 33 Rumyana Neykova, Nobuko Yoshida, and Raymond Hu. Spy: Local Verification of Global Protocols. In Axel Legay and Saddek Bensalem, editors, *Runtime Verification*, volume 8174 of *LNCS*, pages 358–363, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-40787-1_25.
- 34 Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.
- 35 Pine64. PineTime, 2019. URL: <https://www.pine64.org/pinetime/>.
- 36 Jonathan Postel. Rfc0821: Simple mail transfer protocol, 1982.
- 37 Alceste Scalas and Nobuko Yoshida. Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.*, 3(POPL):30:1–30:29, 2019. doi:10.1145/3290343.
- 38 Servo. Servo Web Browser commit, 2015. URL: <https://github.com/servo/servo/commit/434a5f1d8b7fa3e2abd36d832f16381337885e3d>.
- 39 Developers Rust of the library Time. Module std::time documentation, 2023. URL: <https://doc.rust-lang.org/std/time/index.html>.
- 40 Malte Viering, Raymond Hu, Patrick Eugster, and Lukasz Ziarek. A multiparty session typing discipline for fault-tolerant event-driven distributed programming. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–30, October 2021. doi:10.1145/3485501.
- 41 Lennert Wouters, Eduard Marin, Tomer Ashur, Benedikt Gierlich, and Bart Preneel. Fast, furious and insecure: Passive keyless entry and start systems in modern supercars. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):66–85, 2019. doi:10.13154/TCCHS.V2019.I3.66–85.
- 42 Nobuko Yoshida, Fangyi Zhou, and Francisco Ferreira. Communicating finite state machines and an extensible toolchain for multiparty session types. In Evripidis Bampis and Aris Pagourtzis, editors, *Fundamentals of Computation Theory*, pages 18–35, Cham, 2021. Springer International Publishing.

Taking a Closer Look: An Outlier-Driven Approach to Compilation-Time Optimization

Florian Huemer 

Johannes Kepler University, Linz, Austria

David Leopoldseder 

Oracle Labs, Vienna, Austria

Aleksandar Prokopeč 

Oracle Labs, Zurich, Switzerland

Raphael Mosaner 

Oracle Labs, Linz, Austria

Hanspeter Mössenböck 

Johannes Kepler University, Linz, Austria

Abstract

Improving compilation time in optimizing compilers is challenging due to their large number of interconnected components. This includes compiler optimizations, compiler tiers, heuristics, and profiling information. Despite this complexity, research in compilation-time optimization is often guided by analyzing metrics of entire program runs, such as the total compilation time and overall memory footprint. This coarse-grained perspective hides relevant information, such as source program functions for which the compiler allocates a lot of memory or compiler optimizations with a high impact on the total compilation time. This leaves high-level metrics as the only reference point for driving optimization design. Consequently, compilation-time regressions in one program function that are obscured by improvements in other functions stay undetected, while the impacts of compiler changes on untouched parts of the compiler are mainly unknown. Furthermore, developers overlook long-standing compiler defects because their high-level metrics do not change over time.

To address these limitations, we propose ICON, a new data-driven approach to compilation-time optimization that breaks up high-level metrics into individual source program functions, compiler optimizations, or even into individual instructions in the compiler source code. Our methodology enables an iterative in-depth compilation-time analysis, focusing on outliers to identify optimization opportunities. We show that outliers, both in terms of time spent in a particular compiler optimization, and in terms of individual compilations that take substantially longer, can reveal potential problems in the compiler implementation. We applied our approach to GraalVM and extracted data for multiple of its language runtimes. We analyzed the resulting data, present the first detailed look into the distribution of compilation time in the GraalVM compiler, a state-of-the-art multi-language compiler, and identified defects that led to regressions in overall compilation time or the compilation time of specific languages. We furthermore designed two optimizations based on the identified outliers that improve compilation time between 2.25% and 9.45%. We believe that our approach can guide compiler developers in finding usually overlooked optimization potential and defects, and focus future research efforts in making compilers more efficient.

2012 ACM Subject Classification General and reference → Performance; General and reference → Measurement; Software and its engineering → Just-in-time compilers; Software and its engineering → Dynamic compilers

Keywords and phrases Compilation time, outliers, dynamic languages, virtual machines, GraalVM, ICON

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.20



© Florian Huemer, David Leopoldseder, Aleksandar Prokopeč, Raphael Mosaner, and Hanspeter Mössenböck;

licensed under Creative Commons License CC-BY 4.0

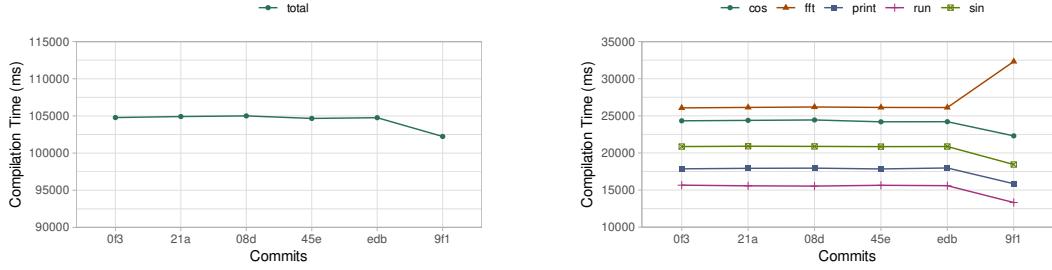
38th European Conference on Object-Oriented Programming (ECOOP 2024).

Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 20; pp. 20:1–20:28



Leibniz International Proceedings in Informatics

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



(a) Results of a hypothetical benchmark measuring total compilation time over the course of several commits. Lower is better.

(b) Results split into individual function compilation times over the course of several commits. Lower is better.

1 Introduction

Improving compilation time in optimizing compilers is challenging due to their large number of interconnected components. Multiple interacting compiler optimizations [24], several compiler tiers, various heuristics, and different profiling information make it challenging to identify the impact of compiler source code changes on the total compilation time of a program. Compilation time should be minimal, especially in modern cloud applications based on serverless functions or microservice architectures that are executed on demand and started frequently [42, 34]. The compilation time for these services is even more relevant for runtimes using just-in-time compilation, where the compilation time directly impacts the startup time of a service [46]. In these runtimes, savings in compilation time are not limited to a single compilation upfront but are reapplied every time an application is started and recompiled. Ahead-of-time-compiled runtimes should also strive for minimal compilation time such as in continuous integration and continuous delivery infrastructures, where compilations take up a large portion of the resource utilization as part of build steps [3].

Despite the high complexity of optimizing compilers and a high demand for fast compilation, research in this field often relies on metrics reflecting entire program runs, such as total compilation time or overall memory footprint, to guide compilation-time optimization. Using fine-grained metrics, such as the time spent compiling individual program functions or the time spent in specific compiler optimizations, could reveal outliers in compilation time that lead to defects and optimization opportunities in the compiler.

To illustrate this problem, consider the hypothetical implementation of a new compiler phase p that introduces an optimization to a compiler. To evaluate the compilation-time impact of p , the conventional approach of analyzing the total compilation time of several benchmarks and comparing it with previous records seems appropriate. As shown in Figure 1a, which illustrates the total compilation time of a hypothetical benchmark over the course of several commits, p , implemented in 9f1, leads to a compilation-time decrease, leaving the impression of a positive compilation-time impact. While this impression seems correct overall, it hides valuable information which is only unveiled by analyzing the compilation time of individual benchmark functions. As shown in Figure 1b, p improves the compilation time of nearly all functions in the benchmark but also results in an outlier with a significant increase in compilation time, the `fft` function. This outlier points towards a defect in p that only occurs under certain conditions, which seem to be present in the `fft` function, and requires further investigation. Consequently, relying solely on metrics of entire program runs to evaluate compilation-time performance can obscure newly-introduced or long-standing defects in compiler implementations.

Therefore, our contribution is to describe a new approach to compiler optimization focusing on outliers to identify defects:

- We describe *Iterative Compilation-time optimization through Outlier-driven Narrowing* (ICON), a novel data-driven approach to compilation-time optimization that splits compilation metrics into individual functions, compiler optimizations, or even into individual instructions in the compiler source code to identify potential problems in compiler implementations by focusing on the outliers in extracted data (Section 3). The approach combines a fine-grained metrics extraction based on iterative narrowing with an outlier-focused approach to finding potential optimization opportunities.
- We present the first detailed look into the distribution of compilation time in the GraalVM compiler, a state-of-the-art multi-language compiler (Section 4). We base our data on the evaluation of 94 benchmarks from the “Are We Fast Yet?”¹ [27], JetStream 2², “Computer Language Benchmark Game,”³ [27] and several internal benchmark suites, analyzing compilation-time metrics for five runtimes, including Python and JavaScript.
- To demonstrate the effectiveness of the ICON approach, we conducted an outlier analysis on the GraalVM compilation-time metrics, identifying one language-agnostic and three language-specific outliers in compilation time (Section 4.5). Through the iterative application of our approach, we narrowed the scopes of the outliers in the compiler and discovered four defects in the GraalVM compiler that were responsible for sub-optimal compilation time in compiler optimizations.
- We sketch the design of two optimizations that target two of the defects identified by our outlier analysis and improve compilation time between 2.25% in Python and 9.45% in Java (Section 5).

2 The Environment of Our Study

We describe ICON as a general methodology to optimize the compilation time of compilers and apply it to a specific runtime environment to emphasize its applicability. We thus begin our technical content by describing GraalVM, the runtime we worked with.

2.1 GraalVM

GraalVM [45] is a state-of-the-art high-performance Java Virtual Machine (JVM) [26] that includes a dynamic optimizing compiler called *GraalVM compiler*. GraalVM provides native support for JVM languages, is implemented in Java [17], and supports just-in-time (JIT) and ahead-of-time (AOT) compilation through the *JVM* [45] and *Native-Image* [42] deployments. The JVM deployment starts programs in the Java interpreter and just-in-time compiles frequently executed methods with the GraalVM compiler, while the Native-Image deployment compiles Java code ahead of time into a native executable, skipping interpretation and compilation of Java code at run time.

In addition to JVM languages, GraalVM supports the execution of guest languages by defining interpreters written with the *Truffle framework* [21]. Truffle is an abstraction layer in GraalVM that provides a domain-specific language API based on Java annotations. It allows to define interpreters and uses *partial evaluation* [44, 15, 9, 28] to transform these interpreters into an intermediate representation that can be further optimized by the GraalVM compiler.

¹ <https://github.com/smarr/are-we-fast-yet>

² <https://browserbench.org/JetStream/in-depth.html>

³ <https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html>

Partial evaluation must be performed at run time in both GraalVM deployments since it requires the input programs of guest language interpreters, which are not known ahead of time. Therefore, in the context of the Truffle framework, both GraalVM deployments support JIT compilation of partially-evaluated guest language interpreters. Consequently, in the Native-Image deployment of a guest language, the interpreter and runtime are AOT compiled, while the user code written in that guest language is still JIT compiled.

Through Truffle and partial evaluation, GraalVM provides official support for seven language runtimes with competitive performance [36, 37] for popular languages such as Python⁴, JavaScript⁵, C/C++ and others via LLVM bitcode⁶ [33], WebAssembly⁷, R⁸, Ruby⁹, and a meta-circular Java runtime called Espresso¹⁰. Apart from implementing different language specifications, these runtimes primarily differ in their internal data structures and interpreter implementations, including abstract-syntax-tree (AST) interpreters, bytecode interpreters, and hybrid approaches that combine aspects of AST and bytecode interpreters.

Our paper focuses on guest language runtimes using Truffle and partial evaluation in the Native-Image deployment of GraalVM. Therefore, the rest of this paper will refer to the Native-Image deployment when talking about GraalVM, the GraalVM compiler, or the Truffle language runtimes.

2.2 GraalVM Compiler

The *GraalVM compiler* [45] is a dynamic optimizing compiler used by GraalVM to compile multiple languages to highly optimized machine code. It contains numerous optimizations that apply platform-specific and platform-independent optimizations [13] and extensively uses *speculative optimizations* [11] based on *assumptions* [38]. If one of the assumptions no longer holds, the GraalVM compiler invalidates the generated machine code and transfers the execution back to the interpreter [38] through *deoptimization* [19].

In the context of guest language interpreters written with Truffle, GraalVM uses the GraalVM compiler as a just-in-time compiler to partially evaluate and compile guest language functions at run time [44]. For this purpose, the GraalVM compiler uses two different configurations called compiler tiers¹¹ [18]. Tier 1 focuses on compilation time and applies fewer short-running optimizations. Tier 2 focuses on optimal machine code and applies all optimizations available in the GraalVM compiler.

The GraalVM compiler’s architecture consists of a *front end*, performing platform-independent compiler optimizations on a high-level intermediate representation (IR) called *GraalIR* [11, 10], and a *back end*, performing register allocation and code generation on a low-level IR called *LIR* [13, 41, 23]. The front end further consists of a *high tier*, *mid tier*, and *low tier*, performing optimizations on different abstraction levels. When compiling guest language functions, the *truffle tier*, performing partial evaluation, precedes the front end.

⁴ <https://github.com/oracle/graalpython>

⁵ <https://github.com/oracle/graaljs>

⁶ <https://github.com/oracle/graal/tree/master/sulong>

⁷ <https://github.com/oracle/graal/tree/master/wasm>

⁸ <https://github.com/oracle/fastr>

⁹ <https://github.com/oracle/truffleruby>

¹⁰ <https://github.com/oracle/graal/tree/master/espresso>

¹¹ <https://docs.oracle.com/javase/8/docs/technotes/guides/vm/performance-enhancements-7.html>

Listing 1 Definition of a scope, timer, and counter with the help of a try-with-resources statement.

```

1 TimerKey timer = DebugContext.timer("Timer");
2 CounterKey counter = DebugContext.counter("Counter");
3
4 void run(Graph graph) {
5     DebugContext debug = graph.getDebugContext();
6     try (DebugContext.Scope scope = debug.scope("Phase")) {
7         DebugCloseable t = timer.start(debug) {
8             ...
9             counter.increment(debug);
10            ...
11        }
12    }

```

2.3 GraalVM Compiler Debug Interface

The *GraalVM compiler debug interface*¹² in the GraalVM compiler enables logging, IR dumping, and the extraction of compilation-time metrics, such as the execution time of the compiler front end, the number of allocated bytes in the compiler mid tier, or the number of generated `mov` instructions during register allocation. The debug interface provides *timers*, for tracking execution time, *memory usage trackers*, for tracking memory usage (i.e., the number of allocated bytes), and *counters*, to keep track of arbitrary counts. This allows developers to get different metric values for parts of the compiler. Developers assign unique names to these metric values and define their measurement scopes with one or several *keys*. Keys allow combining measurements of several compiler regions into one metric value and are combined based on their name. For example, if a compiler performs parts of the same transformation in two different compiler locations and we want to measure the total time of this transformation, we can use two keys with the same name to combine both scopes into the same metric value. A *debug context* stores instances of each metric value. Debug contexts exist once per compilation, and the compiler passes them through all phases. This design enables a per-compilation extraction of metric values.

The debug interface provides *scopes* to enable the logical grouping of keys. Scopes have names and can be nested into one another. The debug context contains helper methods to open scopes while the closing is performed automatically with *try-with-resource* statements, as shown in line 6 of Listing 1. Scopes, timer keys, and memory usage trackers use this automatic closing mechanism, while counter keys need to be manually incremented by the developer, as shown in line 9.

Listing 1 provides an example for the use of the debug interface. When a developer creates a timer key with the help of the `DebugContext` class, as shown in line 1, the call to `timer()` allocates a new key object (`TimerKey`) and links it to a metric value based on the provided name. When the runtime starts the timer in line 7, the key object forwards the name of its metric value to the debug context, which initializes the timer value. After the runtime exits the `try` block in line 11, the timer stops and updates its value in the debug context. At the end of the compilation, the compiler extracts the metric values of all debug contexts and exports them to the console or a file.

¹²<https://github.com/oracle/graal/blob/master/compiler/docs/Debugging.md>

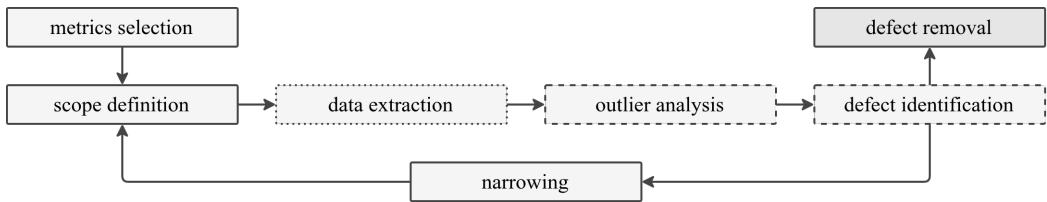


Figure 2 Abstract representation of ICON, consisting of three main phases represented by different border styles.

The GraalVM compiler provides a `Timers` option alongside counterparts for memory usage and counters to enable the extraction of specific metrics. Developers pass these options to the compiler at program startup, which enable metric values based on their name or the name of an enclosing scope. If a key is disabled, the compiler does not extract data for the associated metric value.

3 ICON: Iterative Compilation-Time Optimization Through Outlier-Driven Narrowing

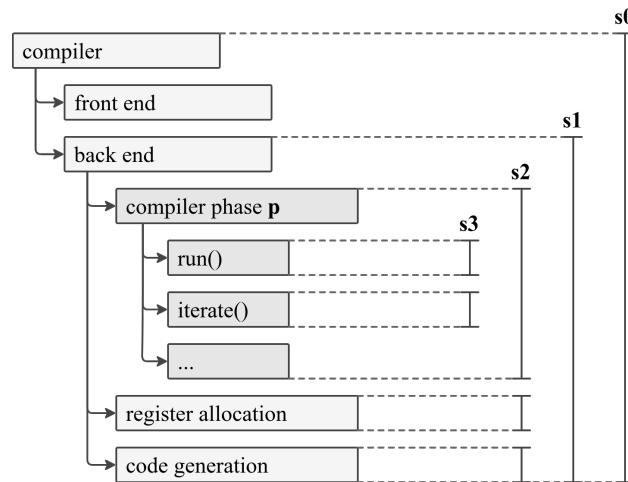
We propose ICON, a new data-driven approach to compilation-time optimization, a methodology focusing on outliers to identify potential problems in compiler implementations. The key idea of ICON is to split up high-level metrics, such as total compilation time or overall memory footprint, into individual *extraction scopes*, such as compiler optimizations, compilations of individual functions, or even into individual instruction in the compiler source code. ICON tries to find defect locations by iteratively narrowing extraction scopes until developers can identify the source of a problem. An outlier analysis after every iteration drives the narrowing process and identifies which extraction scopes to refine. Figure 2 shows an abstract depiction of ICON.

ICON consists of three steps, including the *metric and benchmark selection step* (solid border), the *data extraction step* (dotted border), and the *outlier analysis and defect identification step* (dashed border). To explain the methodology process depicted in Figure 2, consider the hypothetical implementation of a compiler phase p in the compiler back end of an existing compiler. Figure 3 shows a fragment of the resulting compiler architecture.

For the performance evaluation of p , the initial step of ICON is to pick representative benchmarks and metrics, such as compilation time or allocated memory, and to select an initial set of extraction scopes for the first iteration. In our example, the entire compilation pipeline represents a good starting point, as shown by scope s_0 in Figure 3. After applying the compiler code changes and enabling the relevant timers and counters, the next step is to extract a data set based on the selected extraction scopes with the help of the benchmarks. An outlier analysis follows the data extraction to identify outlier compilations via statistical analyses. This outlier analysis results either in a concrete source code location responsible for the outliers or at least in a direction to narrow the selected extraction scopes.

If the outlier analysis points in a new direction for the next iteration, the next step is to select a new set of extraction scopes that narrow the existing ones and to repeat the extraction and analysis process. In our example, a meaningful narrowing is extracting data for the compiler back end, as shown by scope s_1 in Figure 3.

If the data set of the next iteration contains evidence for the previous outliers, the process continues by refining the extraction scopes. Otherwise, the process continues at the previous iteration by narrowing the existing extraction scopes differently. Consequently, since we



■ **Figure 3** Fragment of a compiler pipeline structure after introducing a new compiler phase p.

assume that the narrowing in our example succeeded, the process continues with extraction scopes **s2** searching for outliers in individual compiler phases, including **p**, and scopes **s3**, searching for outliers in individual compiler functions in **p**.

Extraction scopes can theoretically be narrowed down to individual instructions in the compiler source code. The process ends when developers have a clear enough view about the defect locations responsible for the outliers or cannot find any outliers in the data.

While the outlier analysis is essential to ICON, we do not specify a concrete outlier-detection algorithm. Instead, methodology adopters can define concrete algorithms based on the requirements and properties of their compilers. We describe the approach we used for outlier detection in GraalVM in Section 4.4.

Although we focus on a manual inspection and outlier analysis in this paper, an automatic outlier detection and narrowing process could enhance our approach. This automation would aid in integrating our methodology into existing testing and benchmark infrastructures. However, defining such an automated process is outside the scope of our paper.

In the following, we present a detailed definition of *extraction scopes* and describe the necessary changes in GraalVM to support ICON.

3.1 Extraction Scopes

Extraction scopes define the granularity of the data extraction as tuples consisting of a *temporal* and a *spatial* component. The temporal component defines how to split or aggregate the entire execution time of the compiler into individual metric values. Examples are:

- *per execution*. This results in one metric value for the entire run time of the compiler. An example is the extraction of compilation time to compare against historical data.
- *per iteration*. This requires that the program under test executes several iterations and results in one metric value for every iteration. An example is the extraction of compilation time for every benchmark iteration to check their consistency over time.
- *per compilation unit*. This represents the usual approach to metric extraction and results in one metric value for each compilation unit. The metric values depend on the compiler's definition of compilation units (e.g., every function, every module, etc.).
- *per compiler tier*. This is a refinement of *per compilation unit* and results in one metric value per compiler tier. An example is the compilation-time extraction of a compiler phase for every compiler tier to compare the time spent in each compiler tier.

The spatial component defines how to split or aggregate the compilation pipeline (i.e., the compiler source code) into individual metric values. Examples are:

- *per pipeline*. This results in one metric value for the entire compiler pipeline. An example is the extraction of the maximum memory consumption of the compiler when running a benchmark to check that it does not exceed a predefined threshold.
- *per compiler phase*. This results in one metric value for each selected phase in the compiler pipeline. An example is the extraction of compilation time per compiler phase to find optimization potential.
- *per function*. This results in one metric value for each selected function in the compiler source code. An example is the memory-consumption extraction of all functions in a phase that allocates too much memory to find the exact allocation location.
- *per instruction*. This results in one metric value for each selected source code instruction. An example is the extraction of memory allocation for selected source code lines in the compiler.

The temporal and spatial components depend on the compiler architecture and might be linked based on the compiler’s design.

3.2 Implementation in GraalVM

GraalVM was already well fitted to support our methodology. However, in terms of data extraction, some critical parts were missing to support a fine-grained metric value extraction.

The initial implementation of the GraalVM compiler debug interface, as explained in Section 2.3, focuses primarily on the extraction of metric values on a per-name basis. Metrics are extracted for every compiler phase in the GraalVM compiler, so the metric values are associated with the names of the compiler phases. This represents a limitation, since multiple executions of the same compiler phase, which is common in the GraalVM compiler, are merged into a single metric value. An example is the *canonicalizer phase*, which transforms guest language constructs into a canonical form and is executed many times throughout a compilation. Even though these phase executions should be treated independently, the initial debug interface implementation merges their metric values.

To remove this limitation, we extended the existing concept of scopes in the GraalVM compiler debug interface to *aggregate*, *unique*, and *singleton* scopes. Aggregate scopes are similar to the existing scope implementation. As their name suggests, they combine the values produced by several scope executions into a single metric value. Unique scopes on the other hand separate the values of individual scope executions into separate metric values by assigning them a unique name. Singleton scopes ignore the nesting of previous scopes and produce a single metric value, regardless of the scopes they are nested in.

Consider the example in Listing 2. Since the timer key `a_t` in line 5 is part of an aggregate scope, all its executions will contribute to the same metric value `A.Timer`, even though the loop is executed three times. The code in line 12 however, will result in three individual metric values `U_1.Timer`, `U_2.Timer`, and `U_3.Timer`, since the timer key `u_t` is inside a unique scope. Although timer keys `s_t` in lines 7 and 14 are nested in the different outer scopes `A` and `U`, all their executions contribute to the same metric value `S.Timer`, since their nearest nesting scope is a singleton scope.

As a result of using these new scope types, it is possible to extract metric values independently for multiple executions of the same compiler phase. This allows a more precise analysis of the time spent in individual compiler phases and supports the fine-grained metric value extraction required for ICON.

Listing 2 Definition of aggregate, unique, and singleton scopes inside a loop.

```

1 void run (Graph graph) {
2     DebugContext debug = graph.getDebugContext();
3     for (int i = 0; i < 3; i++) {
4         try (DebugContext.Scope a = debug.aggregateScope("A");
5              TimerKey a_t = a.timer("Timer").start(debug)) {
6             try (DebugContext.Scope s = debug.singletonScope("S");
7                  TimerKey s_t = s.timer("Timer").start(debug)) {
8                 ...
9             }
10        }
11        try (DebugContext.Scope u = debug.uniqueScope("U");
12             TimerKey u_t = u.timer("Timer").start(debug)) {
13            try (DebugContext.Scope s = debug.singletonScope("S");
14                 TimerKey s_t = s.timer("Timer").start(debug)) {
15                 ...
16             }
17        }
18        ...

```

4 GraalVM Compilation-Time Evaluation

Based on ICON introduced in Section 3, we extracted compiler metrics for the GraalVM compiler. While our approach primarily focuses on finding defects in compiler implementations, its implementation in GraalVM also enables a detailed view of the distribution of compiler metrics across different compiler phases in the GraalVM compiler. We extracted compilation time and memory usage for all compiler phases based on a large set of benchmarks. In the context of this paper, we will focus on the evaluation of compilation time. The extracted data contains metric values for five partial-evaluation-based language runtimes on GraalVM and allows us to present the first detailed look into the distribution of compilation time across compilation phases in GraalVM. This distribution represents a good starting point for our evaluation and drives the narrowing required for the outlier analysis in Section 4.5.

We start this section by introducing the set of language runtimes we chose to get a representative data set of compilation-time metrics, then proceed by describing the used benchmarks and setup. We conclude by explaining the data extraction process, presenting our results, and performing an outlier analysis on the data.

4.1 Languages

We chose a set of five language runtimes based on the Truffle framework to get a representative data set of the compilation-time distribution in partial-evaluation-based languages compiled by the GraalVM compiler. The runtimes differ in their interpreter implementation and the type system of their implemented languages. Both impact the compilation-time distribution in the GraalVM compiler. Based on the available interpreter implementations and type systems, we chose GraalJS¹³, GraalPy¹⁴, Espresso¹⁵, GraalWasm¹⁶, and the GraalVM LLVM Runtime¹⁷, as further described in Table 1.

¹³ <https://github.com/oracle/graaljs>

¹⁴ <https://github.com/oracle/graalpython>

¹⁵ <https://github.com/oracle/graal/tree/master/espresso>

¹⁶ <https://github.com/oracle/graal/tree/master/wasm>

¹⁷ <https://github.com/oracle/graal/tree/sulong> [33]

 **Table 1** GraalVM language runtimes used for the extraction of compilation-time data.

Runtime	Language	Type system	Interpreter
GRAALJS	JavaScript	dynamic, weak typing	AST interpreter
GRAALPY	Python	dynamic, strong typing	bytecode interpreter
ESPRESSO	Java	static, strong typing	bytecode interpreter
GRAALWASM	WebAssembly	static, strong typing	bytecode interpreter
GRAALVM LLVM RUNTIME	LLVM bitcode	static, strong typing	hybrid interpreter (combines aspects of AST and bytecode interpreters)

4.2 Benchmarks and Setup

We used a set of 94 benchmarks from the “Are We Fast Yet?”¹⁸ [27], JetStream 2¹⁹, “Computer Language Benchmark Game,”²⁰ and several internal benchmark suites for our evaluation. The benchmarks represent real-world computing tasks and are already used internally by different GraalVM language teams to evaluate peak performance, interpreter speed, and memory usage of their language runtimes.

We used “Are We Fast Yet?” for evaluating Espresso, the Java runtime implemented in Truffle. It contains 14 benchmarks focusing on several computing areas, such as JSON string parsing, the computation of the Mandelbrot set, and physics simulations [27].

We used JetStream 2 for evaluating GraalJS and executed 15 of the 64 available benchmarks. These focus on cryptography, physics simulations, or PDF processing. Most excluded benchmarks require browser-specific functionality unavailable in standalone JavaScript engines or WebAssembly support. Since we measured our WebAssembly runtime separately, we did not want to pollute GraalJS results with WebAssembly compilations.

We used the “Computer Language Benchmark Game” [27] benchmarks in combination with other open source benchmarks (*bzip2*²¹, *gzip*²², *stockfish*²³, *oggenc*²⁴) to evaluate GraalPy, the Python runtime, and the GraalVM LLVM runtime. Based on the language runtime, we selected a subset of the available benchmarks used by the respective GraalVM language teams²⁵. Consequently, we used 34 benchmarks for evaluating GraalPy, and 18 for the GraalVM LLVM runtime. The benchmarks include compression algorithms, chess simulations, physics simulations, and scheduling tasks. We compiled the GraalVM-LLVM-runtime benchmarks with a custom LLVM toolchain²⁶ based on the *Clang*²⁷ compiler 16.0.1.

For evaluating GraalWasm, we used an internal benchmark suite consisting of 13 benchmarks. These include the DIGITRON benchmark, an AST interpreter for arithmetic expressions, the FFT [4] benchmark, computing the Fast Fourier transform, and the PHONG [31]

¹⁸ <https://github.com/smarr/are-we-fast-yet>

¹⁹ <https://browserbench.org/JetStream/in-depth.html>

²⁰ <https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html> [27]

²¹ <https://sourceware.org/bzip2/>

²² <https://www.gzip.org/>

²³ <https://github.com/official-stockfish/Stockfish>

²⁴ <https://xiph.org/ogg/>

²⁵ <https://github.com/oracle/graalpython/tree/master/graalpython/com.oracle.graal.python.benchmarks>

²⁶ <https://github.com/oracle/graal/blob/master/sulong/docs/contributor/TOOLCHAIN.md>

²⁷ <https://clang.llvm.org/>

benchmark, computing a shading model for a 3D scene. All of them are written in C and compiled with the *WASI SDK*²⁸ version 21.0 based on the *Clang* compiler version 17.0.0. The benchmark source code is publicly available as part of the GraalWasm repository²⁹.

We executed all benchmarks on the Community Edition (CE) and the Enterprise Edition (EE) of GraalVM. We conducted the execution on an Intel Core i7-8750H with six cores at a fixed CPU frequency of 2.30 GHz and turbo boost disabled. The system has 32GB of main memory and runs Fedora Linux 38 (Workstation Edition).

We compiled all runtimes with *LabsJDK CE 21*³⁰ (`labsjdk-ce-21-jvmci-23.1-b22`). The GraalVM compiler performs individual compilations in separate threads with a separate memory space. We used sufficient iterations to ensure that the GraalVM compiler compiled all relevant functions in each benchmark and ran each benchmark 6 times in separate processes. We used geometric means across all 6 runs to account for measurement inaccuracies due to garbage collection and non-deterministic optimization phases. The resulting numbers represent the evaluation of the Native-Image deployment of GraalVM CE and EE.

4.3 Data Extraction

We surrounded the `run` method, the entry point of each compiler phase, of all compiler phases in the GraalVM compiler with a unique scope, as introduced in Section 3.2, to extract compilation-time data. This implementation allows us to get an individual metric value per phase execution. We put *method inlining* and the *graph decoding* performed during partial evaluation into aggregate scopes since we are not interested in inlining and decoding metrics of individual functions but in their overall impact.

For our evaluation, we extracted the compilation time of every extraction scope in microseconds and computed the compilation time relative to the total compilation time. This normalization is necessary to account for the skewness of the raw data caused by the overrepresentation of short-running compilations in the data set. We extracted metric values for all compilations in our benchmarks and used the compilations of those source program functions that appeared in at least 3 of the 6 benchmark runs to compute the total compilation time for our evaluation. This preselection is necessary to avoid polluting the data with one-time compilations that might represent outliers due to garbage collection pauses or other non-deterministic influences happening exactly during this one compilation.

4.4 Evaluation

We start our evaluation with a high-level overview of the compilation-time distribution in GraalVM. The selected extraction scopes reflect the GraalVM compiler architecture and consist of *partial evaluation*, *compilation*, and *code installation*. We further divide the compilation into a *front end* and *back end*, and the front end into *high tier*, *mid tier*, and *low tier*. We chose the selected granularity to keep the resulting plots clear and readable. In the context of our plots, we define *total compilation time* as the sum of partial evaluation, compilation, and code installation.

²⁸ <https://github.com/WebAssembly/wasi-sdk>

²⁹ <https://github.com/oracle/graal/tree/master/wasm/src/org.graalvm.wasm.benchcases/src/bench>

³⁰ <https://github.com/graalvm/labs-openjdk-21>

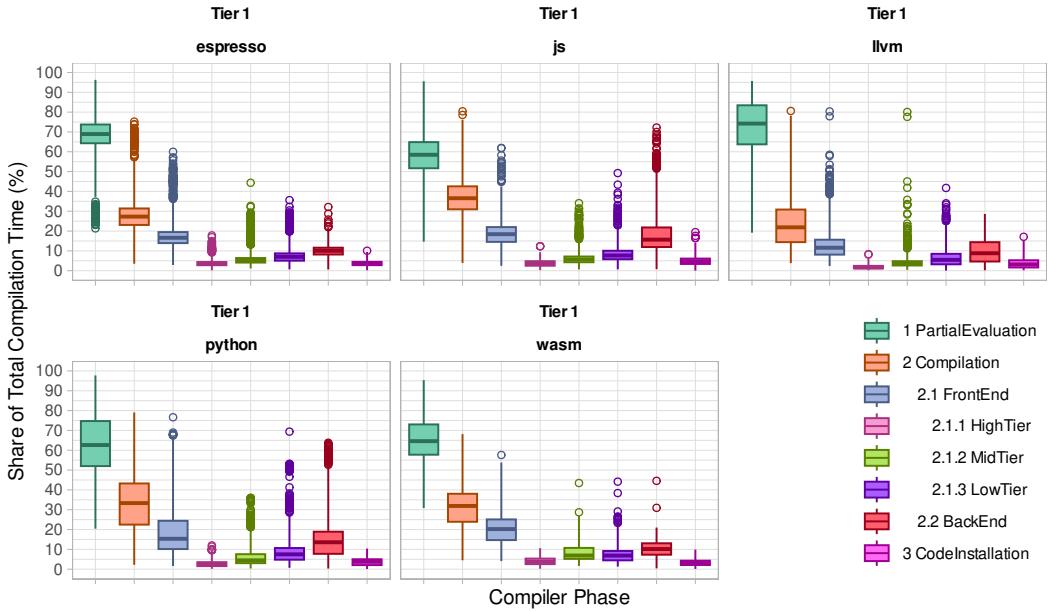


Figure 4 Compilation-time distribution of Tier-1 compilations in GraalVM CE and EE. Compiler phases are from left to right based on their index.

We show the results of the first extraction step in Figure 4, showing the compilation-time distribution of Tier-1 compilations in GraalVM CE and EE. Since their configuration is identical in Tier 1, we combined both editions into a single plot. Figure 5 shows GraalVM-CE Tier-2 compilations, and Figure 6 shows GraalVM-EE Tier 2. The plots show the selected extraction scopes from left to right based on their execution point in the compilations.

Figure 4 shows that partial evaluation has the highest impact on the total compilation time in Tier-1 compilations, with a median between 58.5% and 72.7%. It is also apparent that the median front-end time, in the range of 11.60% to 20.20%, is higher than the back-end time, 8.81% to 15.70%, while the average low-tier time, 5.46% to 7.49%, is higher than the high-tier, 1.66% to 3.79%, and mid-tier times, 3.41% to 6.90%.

Figure 5 shows that the difference in average time between front end and back end is larger in Tier-2 compilations, 7.88%pt (percentage points) to 15.10%pt, than in Tier 1, 1.70%pt to 10.05%pt. This observation seems reasonable since the back-end compiler phases are nearly identical in Tier-1 and Tier-2 compilations, whereas the GraalVM compiler applies additional, longer-running optimization phases in the front end of Tier 2. We verified this conclusion based on absolute numbers to make sure the back-end time stays consistent while the front-end time increases. The only exception is WebAssembly, where the difference between front end and back end is larger in Tier 1 (10.05%pt) than in Tier-2 CE (7.83%pt).

The difference between Tier-1 and Tier-2 front-end and back-end times becomes even more apparent in Figure 6, since Tier-2 EE, with differences in the range of 22.36%pt to 29.11%pt, has additional optimization phases compared to Tier-2 CE. Furthermore, compilation takes an equal amount of time or longer than partial evaluation in Python, with 48.71% and 49.19%, and JavaScript, 50.27% and 46.33%. This can again be explained by the increase in front-end time.

We try to identify possible outliers that require further investigation based on the extracted data. We define *outliers* based on the *interquartile range method* [16] and focus on *extreme* outliers that are more than 3 interquartile ranges (IQR) away from the first (Q1) and third quantile (Q3). We represent outliers in the plots by circles above and below the boxplots.

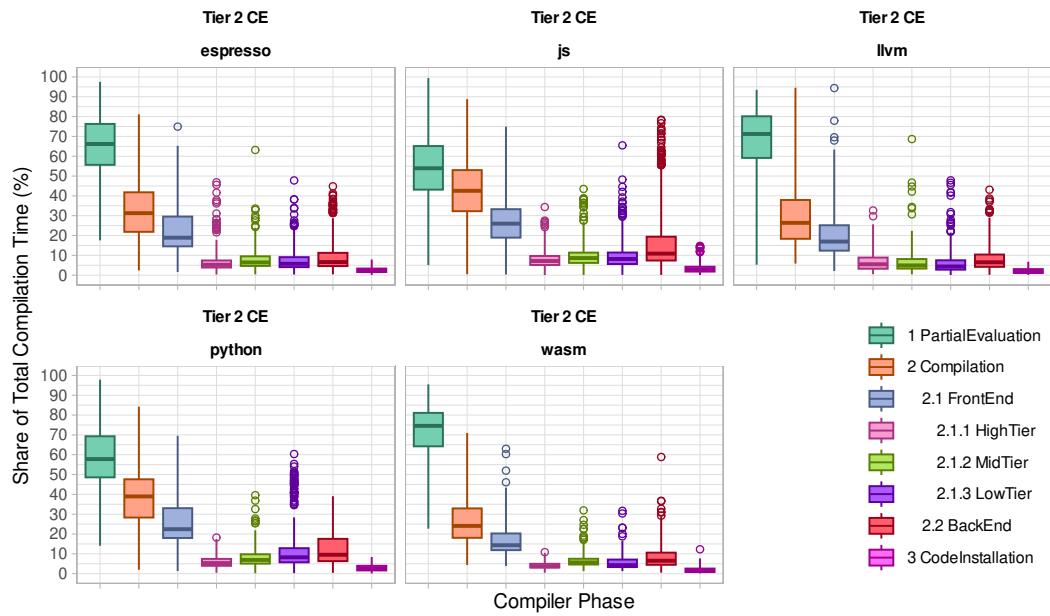


Figure 5 Compilation-time distribution of Tier-2 compilations in GraalVM CE. Compiler phases are from left to right based on their index.

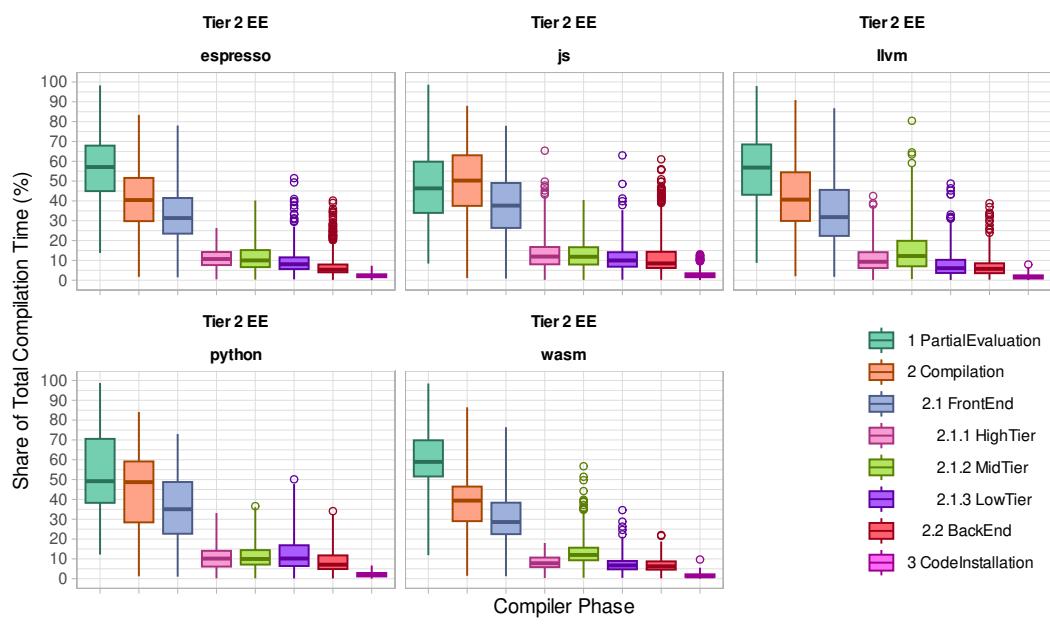


Figure 6 Compilation-time distribution of Tier-2 compilations in GraalVM EE. Compiler phases are from left to right based on their index.

All three data sets contain many outliers within individual extraction scopes. For example, Espresso and JavaScript contain a lot of outliers in all extraction scopes of Tier-1 compilations, while the GraalVM LLVM runtime shows significant outliers in the mid tier. In Tier-2 compilations, JavaScript presents significant outliers in the back end, while Python contains several outliers in the low tier of CE compilations. The GraalVM-LLVM-runtime outliers found in the mid tier of Tier-1 compilations are also found in Tier 2.

Analyzing all outliers in all data sets would be possible. However, we will focus on the most significant outliers to identify defects and optimizations with a high impact potential. To select the relevant outliers, we prioritize them based on their distance from Q_1 or Q_3 in terms of multiples of the IQR and look into extraction scopes with a high outlier density.

Consequently, we will analyze the outliers in the mid tier of GraalVM-LLVM-runtime compilations. Especially the outliers in Tier-1 compilations have a distance of more than $30IQR$ from Q_3 . The outliers are consistent throughout all editions and compiler tiers and are responsible for up to 80% of the total compilation time in Tier 1 and Tier-2 EE.

Furthermore, we will analyze the back end of JavaScript compilations. This extraction scope shows a high outlier density throughout all editions and compiler tiers. The same applies to the low tier of Python Tier 1 and Tier-2 CE compilations.

Although it is hard to quantify partial evaluation as an outlier, it is apparent from a visual analysis of the plots in Figures 4 to 6 that partial evaluation takes up a significant part of the total compilation time in all compiler editions and tiers. Therefore, in addition to our outlier analyses, we show the applicability of our methodology for finding optimizations in existing compiler phases by analyzing partial evaluation in all guest language runtimes.

4.5 Outlier Analysis

We perform an outlier analysis based on the possible defects identified during the evaluation of the GraalVM compilation time in Section 4.4. We verify the defects and identify their source code locations. In addition, we show that our approach can find optimization locations in compilers by focusing on long-running extraction scopes.

4.5.1 Mid Tier in the GraalVM LLVM Runtime

We first analyzed the outliers in the GraalVM-LLVM-runtime mid tier and identified the benchmark functions responsible for the outliers to find a possible defect in the compilations. The outliers were two functions in the *bzip2* and *oggenc* benchmarks that did not show any suspicious characteristics in the C source code. Next, we refined our extraction scopes from a top-level view of the mid tier to individual compiler phases. Figure 7 shows the result of the narrowed extraction scopes of Tier-1 compilations. The figure shows that the *frame-state-assignment phase* has a lot of outliers and the previously identified functions indeed spend nearly all of their mid-tier time in the frame-state-assignment phase of Tier-1 compilations. We also verified that the frame-state-assignment phase is the defect source in Tier-2 compilations (omitted from the paper).

Frame states are a mapping from machine state (i.e., registers and native stack frames) to interpreter state (i.e., JVM stack frames) and are required by GraalVM during deoptimizations to regenerate the interpreter state of a program [11, 12]. During partial evaluation, the GraalVM compiler generates a *frame-state* node for every operation that changes the local state of a method (local variables, operand stack values, and locked objects) and attaches it to the node in the GraalIR graph that causes this change. Subsequent compiler optimization phases might also introduce new nodes, and therefore new frame states. When entering

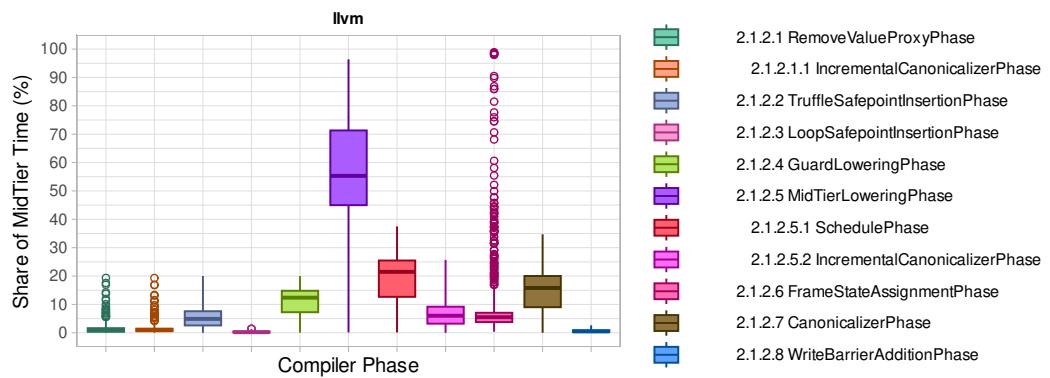


Figure 7 Compilation-time distribution of the mid tier in Tier-1 compilations. Compiler phases are from left to right based on their index.

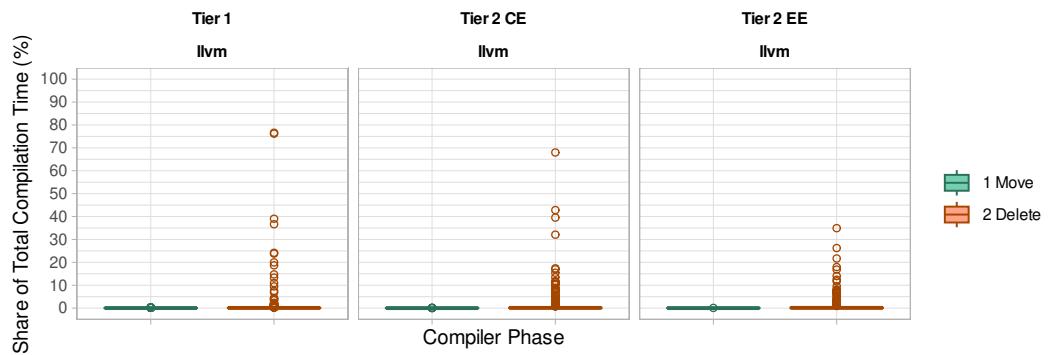


Figure 8 Compilation-time distribution of the two frame-state-assignment phase parts Move and Delete. Compiler phases are from left to right based on their index.

the frame-state-assignment phase, the graph is stable and no more nodes that may cause deoptimizations can be introduced in later compiler phases. Since the frame states are only required by deoptimizations, the frame-state-assignment phase moves the frame states from their originating nodes to deoptimization nodes at deoptimization points in the graph and removes unused frame states.

To further narrow the defect location, we introduced individual timers for the two source code method calls (*Move* and *Delete*) in the `run` method of the frame-state-assignment phase. The *Move* method moves existing frame-state nodes, while the *Delete* method deletes unused frame-state nodes. Figure 8 shows the result of this last narrowing step. The values of these last extraction scopes are relative to the total compilation time. The figure shows that the deletion of unused frame-state nodes is responsible for the outliers and should be optimized, as described in Section 5.1.

4.5.2 Back End in JavaScript

We analyzed the outliers in the JavaScript back end and identified that all functions responsible for the outliers are part of the *typescript* benchmark, which compiles a large TypeScript application to JavaScript. The benchmark uses the identified functions to walk the AST of the TypeScript application. The functions themselves are small and call each other recursively. As a result of these recursive calls, the GraalVM compiler can perform a lot of inlining.

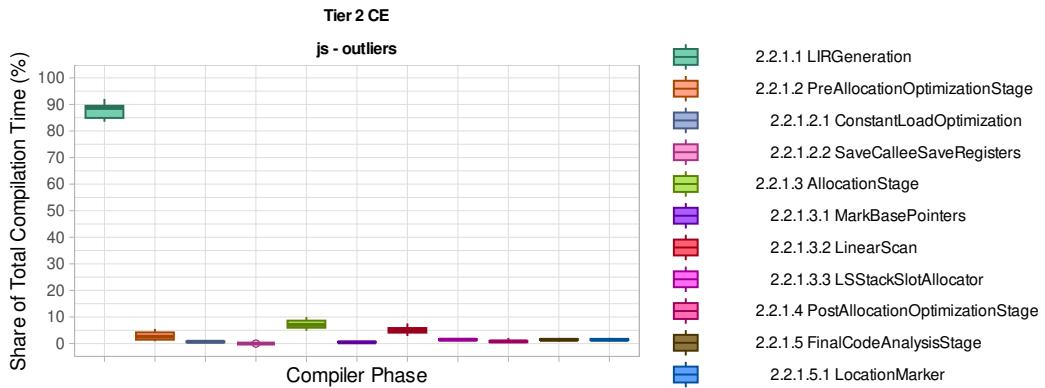


Figure 9 Compilation-time distribution of the back end in Tier-2 CE compilations. Compiler phases are from left to right based on their index.

We narrowed the extraction scopes to individual compiler phases in the back end and found that the identified functions on average spend 87% of their Tier-2 CE back-end time in *LIR generation*, as shown in Figure 9. Tier-1 (60%) and Tier-2 EE (82%) compilations produced similar numbers. The LIR-generation phase transforms high-level GraalIR nodes into low-level LIR nodes by iterating over the GraalIR graph and transforming each node.

We found out that the outlier functions spend most of their time transforming nodes that generate a *LIR frame state* during LIR generation. The LIR frame states represent garbage collection and deoptimization information and the GraalVM compiler computes them based on the frame-state nodes in the GraalIR graph by iterating over all values in the frame state and all its parent frame states and transforming the GraalIR representation of the frame-state values into LIR representations. A frame state has a parent frame state if its method was inlined into another method. The parent frame state represents the frame state of the method into which this method was inlined. We confirmed that the outlier functions had a lot of frame states as a result of inlining and, through a last narrowing step, confirmed that the identified outliers spend most of their time generating LIR frame states.

Optimizing this pattern would require significant changes in the compiler architecture which was not in the scope of this paper. We reported our findings to the GraalVM compiler team for further investigation.

4.5.3 Low Tier in Python

We analyzed the outliers in the Python low tier and found out that all outliers are instances of the same function in all benchmarks, the *time* function in the Python *time* module³¹. This function is built into GraalPy and returns the current time in seconds. The benchmarks use this function to measure execution time.

We narrowed the extraction scopes to individual compiler phases in the low tier and found that the identified functions spend more than 95% of the low-tier time in the *low-tier-lowering phase*. Lowering transforms nodes on a higher abstraction level to node structures on a lower abstraction level. For example, a *load-field* node, accessing a field on an object, is lowered to a *read* node, reading a value from an address in memory.

³¹ <https://docs.python.org/3/library/time.html#time.time>

We also extracted the node count for the graph of the *time* function and found that it is a small graph with only 157 nodes. We narrowed the extraction scopes to the lowering of individual node types and found that four individual nodes were responsible for the time spent in lowering. These were two *exception-object* nodes, one *truffle-safepoint* node, and one *new-instance* node. The commonality between these nodes is that they are all lowered with the help of *snippets* [35]. The GraalVM compiler creates these snippets the first time they are used and caches them based on their compilation unit. This implies an initial overhead that is, in most cases, amortized by several usages of the cached snippets. However, since the graphs for the *time* functions only contain one or two of the snippet-lowered nodes, this results in a significant run-time overhead during the low-tier-lowering phase.

Optimizing this process would again require significant changes to the implementation of snippets and the lowering process itself, which is out of the scope of this paper. We reported our findings to the GraalVM compiler and GraalPy teams for further investigation.

4.5.4 Partial Evaluation

The partial-evaluation time of compilations is a long-standing problem in the GraalVM compiler and, as our evaluation shows, is responsible for up to 73% of the total compilation time. Therefore, finding an optimization opportunity in partial evaluation would improve a large portion of the overall compilation time in GraalVM guest language interpreters.

We narrowed the extraction scopes to sub-phases of partial evaluation to find possible optimizations to partial evaluation. The sub-phases include *decode-graph*, *cleanup-graph*, two *post-partial-evaluation suites*, *decode-inline-graph*, and the *inline post-partial-evaluation suite*, apart from several other optimization phases and sub-phases. Figure 10 shows the result of the narrowed extracted scopes. The plots only show the sub-phases with the highest compilation-time impact to improve readability.

Figure 10 shows that *graph decoding* (*DecodeGraph*, *DecodeInlinedGraph*) has the highest impact on compilation time spent in partial evaluation. Graph decoding iterates over a graph representation of the program IR, processing one node after the other. During the decoding, the GraalVM compiler applies several optimizations based on the type of each node. *Load-field* nodes for example, can be constant folded, while *invoke-with-exception* nodes, representing function calls, can be inlined. Partial evaluation performs graph decoding on the main partial-evaluation function (*DecodeGraph*), the function for which Truffle requests partial evaluation, and all functions inlined during partial evaluation (*DecodeInlinedGraph*). In Tier-2 compilations, graph decoding of inlined functions dominates the partial-evaluation time due to aggressive inlining policies, while in Tier 1, inlining is restricted.

We extracted the time spent processing specific node types during graph decoding by adding singleton scopes for all node types, as described in Section 3.2. Examples of node types are *load-field* nodes that load the field of an object, *if* nodes that represent an if statement, and *loop-begin* nodes representing a loop header. Figure 11 shows the result of the individual node types. The plots only show the node types with the highest impact on compilation time to improve readability.

Figure 11 shows that *invoke-with-exception* nodes dominate the graph-decoding process. Since *invoke-with-exception* nodes perform function inlining, it is expected that these nodes take up most of the total graph-decoding time. During inlining, *invoke-with-exception* nodes have to recursively decode the callee function and attach the resulting graph to the graph that is currently decoded. This process is already heavily optimized and most of its time is spent in the recursive decoding. Therefore, we did not further consider *invoke-with-exception* nodes for finding optimization potential. We instead focused on *load-field* nodes. We optimized the constant-folding performed for *load-field* nodes to improve the graph-decoding performance, as described in Section 5.2.

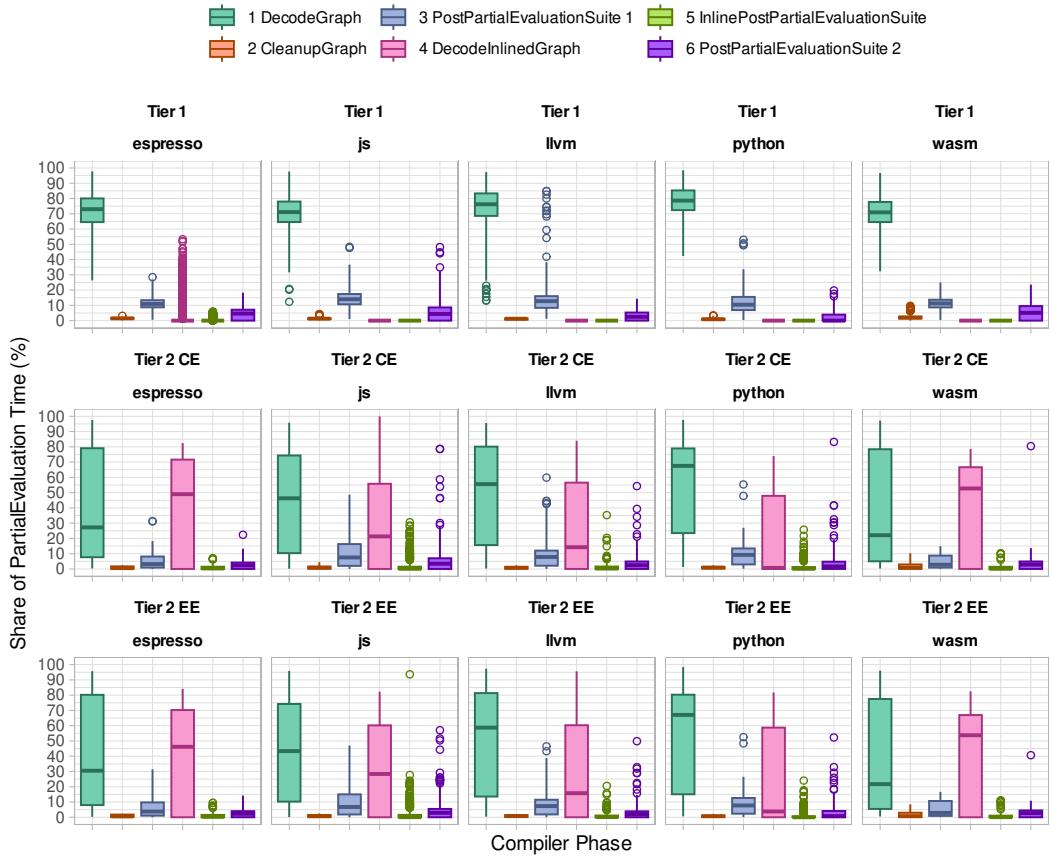


Figure 10 Compilation-time distribution of partial evaluation. Compiler phases are from left to right based on their index.

5 Optimization

5.1 Frame State Assignment – Optimizing Deletion Strategies

Graphs are a common way of implementing the intermediate representation of a compiler [39, 8, 7]. The design of these graphs is crucial because optimizations require an efficient way of traversing and manipulating the IR. The efficiency of these operations depends on the data structures used to represent nodes and edges in the graph.

GraalVM uses a directed graph that represents data flow and control flow in a single data structure [11, 12]. To model data flow, a node has *input edges*, pointing from the node using a value to the node defining this value. The reverse edges, so-called *usage edges*, which point in the opposite direction, are automatically maintained by the graph, so optimizations do not have to deal with maintaining them. However, in contrast to input edges, usage edges are not ordered and can only be accessed as an unordered set [11, 12]. Consequently, finding a specific usage of a node may require traversing the entire set. Unordered sets are a common way of reducing the memory footprint of the reverse-edge sets in JIT compilers³².

³²<https://chromium.googlesource.com/v8/v8/+refs/heads/main/src/compiler/node.h#181>,
<https://github.com/openjdk/jdk/blob/master/src/hotspot/share/opto/node.hpp#L319>

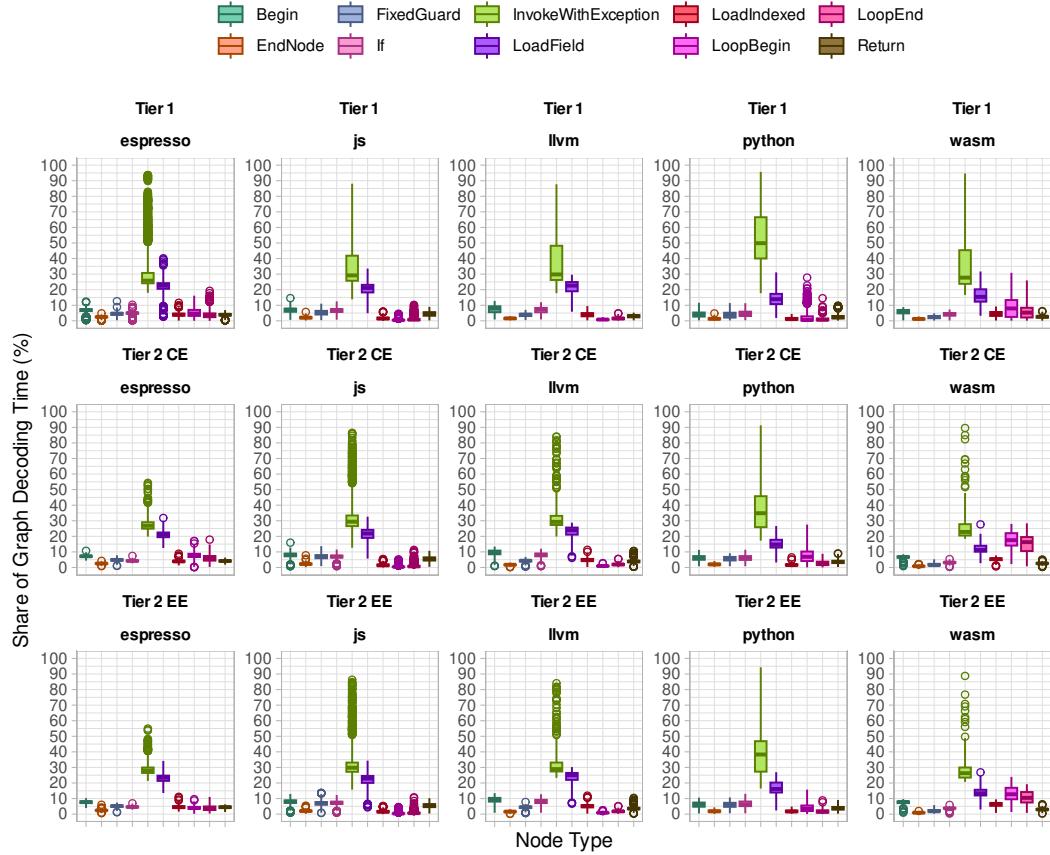


Figure 11 Compilation-time distribution of individual node types during graph decoding. Node types are from left to right in alphabetical order.

Usage edges are needed, for example, for deleting unused nodes in dead code elimination [29]. If a node is no longer used, its usage edges must be removed from all its inputs before the node can be deleted. As a result of deleting a usage of an input node, the input node itself can become unused, i.e., its usage set may become empty. This can lead to the transitive deletion of other nodes, which, in the worst case, requires several full traversals of potentially countless node usage sets.

For the frame-state-assignment phase of the GraalVM compiler, the outlier analysis in Section 4.5.1 showed that the deletion of unused frame-state nodes is responsible for a substantial part of the mid-tier time.

Normally, the traversal of usage sets during the deletion of frame-state nodes does not represent a problem, since the number of frame-state nodes is usually small. However, the graphs in which we identified outliers contained hundreds of frame-state nodes due to excessive inlining, resulting in slowdowns in the frame-state-assignment phase. Therefore, we updated the algorithm for deleting nodes. In the original implementation, every unused node was visited one after the other, and the usage sets of the node's inputs were updated immediately when deleting the node. This led to a lot of usage set traversals due to the transitive deletion of nodes.

Instead of updating the usage sets of nodes immediately, the updated algorithm tracks the usage counts of nodes in a separate map. The algorithm performs a depth-first traversal of the graph starting at the inputs of the deleted frame-state nodes, updates the usage counts

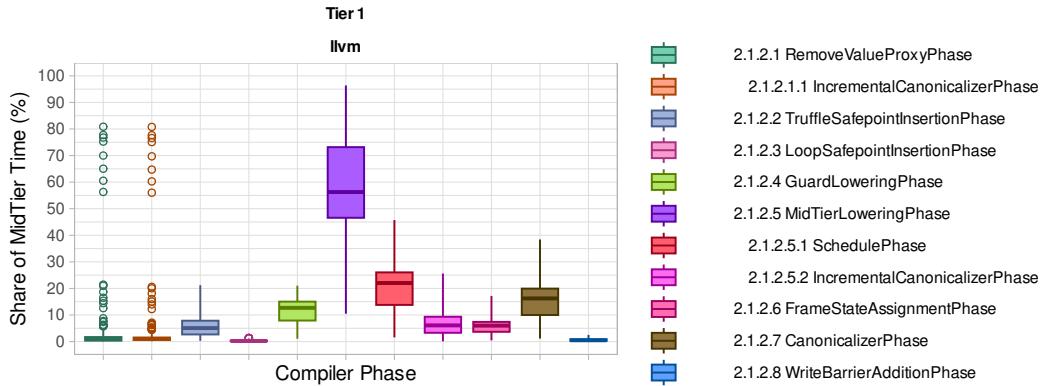


Figure 12 Compilation-time distribution of the mid tier in Tier-1 compilations after optimizing the frame-state-assignment phase. Compiler phases are from left to right based on their index.

in the map, and deletes unused nodes along the way. The traversal stops at nodes whose usage count in the map is non-zero. After the graph traversal, only nodes that are alive and have changed usage counts need to be updated. The pseudocode for the updated algorithm can be found in the appendix (Listing 3).

As a result of this optimized algorithm, the previous outliers no longer exist. We show the updated mid-tier compilation time of Tier-1 compilations in Figure 12. We also verified that the outliers no longer exist in Tier 2 (omitted from the paper). Due to the reduction of the time spent in the frame-state-assignment phase, new outliers in the *incremental canonicalizer phase* (2.1.2.1.1) arose that would be worth investigating, but are outside the scope of this paper. We confirmed that the new outliers were not caused by changes in the frame-state-assignment phase but were already present in the previous data set.

5.2 Graph Decoding – Constant-Fold Caches

Constant-folding is a crucial operation in compilers based on partial evaluation, because it allows these compilers to evaluate parts of the program at compilation time, and thus simplify the program [29, 1, 40]. When a partial-evaluation-based compiler identifies a constant in the program representation, many subsequent operations depending on that constant can also be replaced with constants. For example, a read operation from a field of a constant object (i.e., an object represented by a constant pointer), can be evaluated during compilation, and can be replaced with another constant that holds the value of the respective field.

Object-field reads and array reads on constant values are very common operations in interpreters, because the program representation is encoded in either ASTs or bytecode arrays [25], and the interpreter is partially evaluated for a given section of the program that is a constant value from the perspective of the compilation. Thus it is critical that constant-folding is executed very efficiently.

Reading a field from a constant object requires the compiler to (1) determine whether the field or an array location is guaranteed to remain constant after partial evaluation, and (2) to read the value from the respective offset in the program's memory. The first step usually relies on modifiers or annotations found in the source code of the interpreter. Both of these steps involve calls to the runtime environment, and rely on reflective metadata, which is usually expensive to obtain compared to a simple object-field read.

In the GraalVM compiler, the outlier analysis in Section 4.5.4 showed that the constant-folding of load-field nodes (which represent object-field reads in GraalIR) is responsible for a substantial part of the partial-evaluation time.

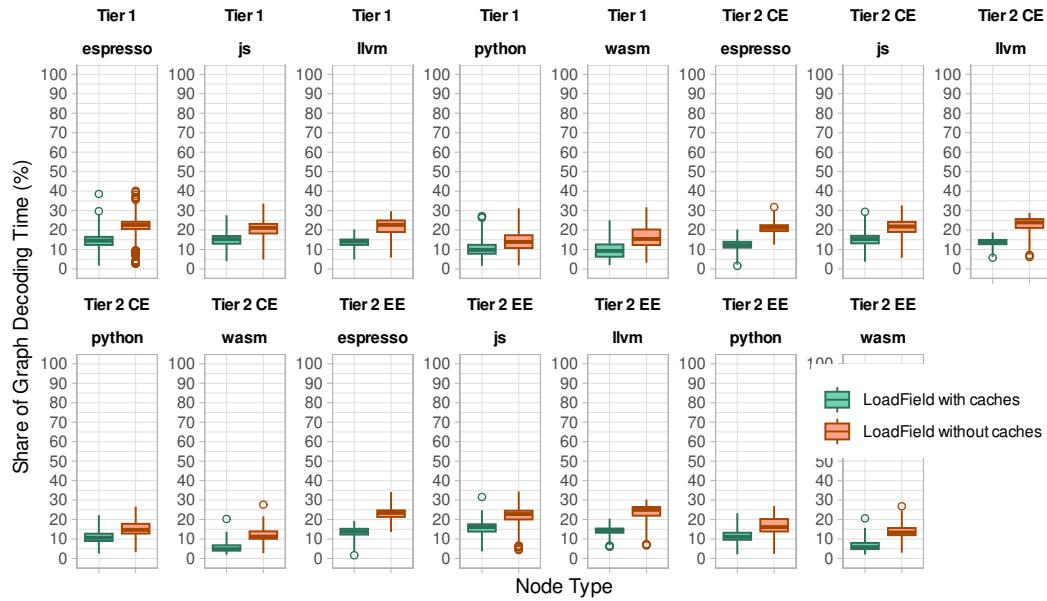


Figure 13 Comparison of time spent in load-field nodes during graph decoding before and after the introduction of caches. The time with caches is on the left in each plot, the time without caches on the right.

If the same constant occurs multiple times in an IR graph, it is represented by a single IR node to reduce memory footprint and compilation time [7, 8, 11, 12]. To maintain this graph property, the GraalVM compiler, before adding a new constant node to the graph, performs a graph traversal to look for equivalent nodes. This can be an expensive operation depending on the graph size.

In the current partial-evaluation implementation in the GraalVM compiler, all constant-folding attempts are independent. This means that the reflective metadata and value are loaded again and again regardless of whether this information was already retrieved by a previous constant-folding attempt. In addition, every constant-folding attempt allocates a new constant node that is discarded when the GraalVM compiler identifies an equivalent node in the graph. On average, the same constant field is read 8.5 times per compilation unit across all compiler tiers and editions. Therefore, to reduce the number of calls to the runtime environment as well as the number of allocated constant nodes and the number of graph traversals, we introduced a two-layer cache system in the constant-folding performed during partial evaluation.

As a result, the impact of constant-folding on partial evaluation was reduced, as shown in Figure 13. Overall, this optimization led to a compilation-time reduction between 2.25% (Python) and 6.88% (LLVM Runtime) in Tier 1, between 2.48% (Python) and 8.16% (WebAssembly) in Tier-2 CE, and between 4.49% (JavaScript) and 9.45% (Espresso) in Tier-2 EE.

6 Related Work

There is extensive research in the field of compilation-time optimization [22, 20, 30, 24, 2]. However, ICON, focusing on identifying outliers to find optimization opportunities in existing compilers, combines multiple aspects that we are not aware of being found together in any

sole research. It combines (1) iterative narrowing of scopes to analyze a problem with (2) focusing on outliers in extracted data to (3) improve compilation-time metrics. We, therefore, focus on related work in compiler optimization similar to ICON in at least one aspect.

Brown et al. [5], propose a data-driven methodology to identify the impact of compiler optimizations on security-oriented aspects of the generated machine code. They evaluate 20 benchmarks and analyze the availability of gadget sets used for code reuse attacks based on the enabled compiler optimizations in the GCC and Clang compilers. They perform a coarse-grained analysis based on optimization levels available in GCC and Clang, and a fine-grained analysis on individual optimizations in those compilers, similar to the narrowing of scopes in ICON. Furthermore, they use an outlier analysis to identify relevant compiler optimizations, similar to the outlier analysis defined in ICON.

Bryksin et al. [6], propose a method to identify code anomalies in order to find issues in compilers. They focus on source code fragments that are not typically found in a given programming language or uncharacteristic bytecode produced by a compiler. With anomaly detection algorithms, similar to the outlier detection in ICON, they identified several optimization opportunities in the Kotlin compiler.

Regarding compilation-time optimization, existing work can be categorized into approaches for phase selection and ordering [22, 30, 24, 2], automatic compiler optimization level selection [20], and automatic compiler heuristics or optimization tuning [14, 32]. Most existing approaches use machine learning and primarily focus on the common case instead of outliers.

6.1 ICON-like Approaches in Other Compilers

Based on online reports³³, compilation time is an important factor for many state-of-the-art compiler implementations. To improve these metrics, some compiler teams are actively working on improving their tooling to extract compilation-time metrics³⁴.

We surveyed the *V8*³⁵ JavaScript and WebAssembly compiler, the *Clang*³⁶ compiler in LLVM, the *GCC*³⁷ compiler, the Java *HotSpot* [23] compiler, and the C# *RyuJIT*³⁸ compiler in order to identify to which extent their capabilities overlap with the ideas proposed by ICON. Table 2 shows our findings from analyzing the documentation and source code of the compilers, as well as information provided by online forums and mailing lists. We tried to find out whether the compilers provide a way of extracting compilation-time metrics (column 1), whether they support the narrowing of extraction scopes (column 2), and whether they try to optimize compilation time based on outliers (column 3).

Based on the available compiler source code and the presence of compiler flags described in the online documentation, all surveyed compilers provide compilation-time metrics, although to varying degrees. To the best of our knowledge, HotSpot provides the compilation time

³³<https://github.com/llvm/llvm-project/labels/slow-compile>,
<https://gcc.gnu.org/pipermail/gcc-bugs/2024-March/857635.html>,
<https://discourse.llvm.org/t/gsoc-2024-statistical-analysis-of-llvm-ir-compilation-with-clang/77532>

³⁴<https://bugs.openjdk.org/browse/JDK-8311896>

³⁵<https://v8.dev/>

³⁶<https://clang.llvm.org/>

³⁷<https://gcc.gnu.org/>

³⁸<https://github.com/dotnet/runtime/blob/main/docs/design/coreclr/jit/ryujit-overview.md>

Table 2 Capabilities of compilation-time optimization of V8, Clang, GCC, HotSpot, RyuJIT, and GraalVM with ICON.

Compiler	Metrics extraction	Scope narrowing	Outlier analysis
V8	yes	no	no
CLANG	yes	yes	no
GCC	yes	yes	no
HOTSPOT	yes	no	no
RYUJIT	yes	no	no
GRAALVM WITH ICON	yes	yes	yes

across all compilations via the `-XX:+CITime` flag. V8 and RyuJIT report a fixed set of metrics for all compilation units via compiler flags (V8³⁹, RyuJIT⁴⁰). Clang⁴¹ and GCC⁴² provide detailed reports about time spent in individual optimizations via the `-ftime-report` flag.

Regarding scope narrowing, Clang allows passing a parameter to the `-ftime-report` compiler flag to differentiate whether the time is reported “per-pass” or “per-pass-run” to separate or combine individual pass executions. GCC supports narrowing via a separate compiler flag `-ftime-report-details`. As far as we know, none of the other compilers provide options to change the scope of extracted metrics.

To the best of our knowledge, the GraalVM compiler with our ICON enhancements is the only compiler using outliers to improve compilation time.

6.2 Synergy with Regression Testing

In addition to finding optimization opportunities in compilers, ICON is well suited to accompany compiler regression testing. While regression testing ensures that changes to the source code do not break existing compiler features and do not impair the correctness of the produced machine code [43], our approach ensures that changes to the source code do not lead to compilation-time regressions, additional memory allocation, or other aspects negatively impacting compilation. Therefore, in addition to ensuring the correctness of the output via regression testing, ICON ensures the efficiency of the compilation process and identifies any newly introduced defects. Both can execute the same tests, so no additional input programs are required to integrate ICON.

7 Conclusion

We presented ICON, a new data-driven approach to compilation-time optimization that splits high-level metrics into individual source program functions, compiler optimizations, or even into individual instruction in the compiler source code. By focusing on outliers in the extracted data, this approach can identify potential optimization opportunities in compiler implementations that are usually overlooked and provides a systematic approach to the analysis of compilation-time metrics.

³⁹ <https://chromium.googlesource.com/v8/v8/+/refs/heads/main/src/diagnostics/compilation-statistics.h>, <https://v8.dev/docs/trace>

⁴⁰ <https://github.com/dotnet/runtime/blob/main/src/coreclr/jit/compiler.h#L1643>, <https://github.com/dotnet/runtime/blob/main/docs/design/coreclr/jit/viewing-jit-dumps.md#miscellaneous-always-available-configuration-options>

⁴¹ <https://releases.llvm.org/12.0.0/tools/clang/docs/ClangCommandLineReference.html#cmdoption-clang1-ftime-report>

⁴² <https://gcc.gnu.org/onlinedocs/gcc/Developer-Options.html#index-ftime-report>

To demonstrate the effectiveness of our approach, we used ICON to extract a detailed view of the compilation time of the individual optimizations of the GraalVM compiler and performed a comprehensive outlier analysis on the resulting data with the goal of finding optimization potential. We found that most of the compilation time is spent in partial evaluation throughout all compiler tiers and editions, while the front end takes more time than the back end, especially in Tier-2 compilations.

The outlier analysis led to one language-agnostic and three language-specific outliers in compilation time. In the outlier analysis, the spatial component of the extraction-scope narrowing turned out to be useful at all levels. While the per-node analysis of partial evaluation helped us to identify an optimization opportunity in constant-folding, the per-phase analysis applied to the compiler mid tier, low tier and back end led to the detection of compiler defects in Python, JavaScript, and the GraalVM LLVM runtime, and helped us to develop an improved algorithm for the frame-state-assignment phase. Similarly, the temporal component was useful for analyzing Python, and we identified a single function as the compiler defect source.

Based on the identified optimization opportunities, we added additional caches to the constant-folding performed during partial evaluation and implemented a new deletion strategy for unused nodes in the frame-state-assignment phase. We reported the two remaining findings to the GraalVM compiler and language teams. The implemented optimizations improved compilation time in all languages between 2.25% (Python) and 9.45% (Espresso).

References

- 1 Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986. URL: <https://www.worldcat.org/oclc/12285707>.
- 2 Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. A survey on compiler autotuning using machine learning. *ACM Comput. Surv.*, 51(5):96:1–96:42, September 2019. doi:[10.1145/3197978](https://doi.org/10.1145/3197978).
- 3 Islem Bouzenia and Michael Pradel. Resource usage and optimization opportunities in workflows of github actions. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 25:1–25:12, Los Alamitos, CA, USA, April 2024. ACM. doi:[10.1145/3597503.3623303](https://doi.org/10.1145/3597503.3623303).
- 4 E. O. Brigham and R. E. Marrow. The fast fourier transform. *IEEE Spectrum*, 4(12):63–70, 1967. doi:[10.1109/MSPEC.1967.5217220](https://doi.org/10.1109/MSPEC.1967.5217220).
- 5 Michael D. Brown, Matthew Pruitt, Robert Bigelow, Girish Mururu, and Santosh Pande. Not so fast: understanding and mitigating negative impacts of compiler optimizations on code reuse gadget sets. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–30, October 2021. doi:[10.1145/3485531](https://doi.org/10.1145/3485531).
- 6 Timofey Bryksin, Victor Petukhov, Ilya Alexin, Stanislav Prikhodko, Alexey Shpilman, Vladimir Kovalenko, and Nikita Povarov. Using large-scale anomaly detection on code to improve kotlin compiler. In Sunghun Kim, Georgios Gousios, Sarah Nadi, and Joseph Hejderup, editors, *MSR ’20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, MSR ’20, pages 455–465, New York, NY, USA, 2020. ACM. doi:[10.1145/3379597.3387447](https://doi.org/10.1145/3379597.3387447).
- 7 Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Trans. Program. Lang. Syst.*, 17(2):181–196, March 1995. doi:[10.1145/201059.201061](https://doi.org/10.1145/201059.201061).
- 8 Cliff Click and Michael Paleczny. A simple graph-based intermediate representation. In Michael D. Ernst, editor, *Proceedings ACM SIGPLAN Workshop on Intermediate Representations (IR’95), San Francisco, CA, USA, January 22, 1995*, IR ’95, pages 35–49, New York, NY, USA, 1995. ACM. doi:[10.1145/202529.202534](https://doi.org/10.1145/202529.202534).

- 9 Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Mary S. Van Deusen and Bernard Lang, editors, *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993, POPL '93*, pages 493–501, New York, NY, USA, 1993. ACM Press. doi:10.1145/158511.158707.
- 10 Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. Graal ir: An extensible declarative intermediate representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, pages 1–9, 2013.
- 11 Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In Christoph Bockisch, Michael Haupt, Steve Blackburn, Hridesh Rajan, and Joseph Gil, editors, *VMIL@SPLASH '13: Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages, Indianapolis, IN, USA, 28 October 2013, VMIL '13*, pages 1–10, New York, NY, USA, 2013. ACM. doi:10.1145/2542142.2542143.
- 12 Gilles Marie Duboscq. Combining speculative optimizations with flexible scheduling of side-effects, 2016. URL: <https://resolver.obvsg.at/urn:nbn:at:at-ubl:1-9708>.
- 13 Josef Eisl. Trace register allocation, 2018. URL: <https://resolver.obvsg.at/urn:nbn:at:at-ubl:1-25787>.
- 14 Grigori Fursin, Yuryi Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, François Bodin, Phil Barnard, Elton Ashton, Edwin V. Bonilla, John Thomson, Christopher K. I. Williams, and Michael F. P. O’Boyle. Milepost GCC: machine learning enabled self-tuning compiler. *Int. J. Parallel Program.*, 39(3):296–327, June 2011. doi:10.1007/s10766-010-0161-2.
- 15 Yoshihiko Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *High. Order Symb. Comput.*, 12(4):381–391, December 1999. doi:10.1023/A:1010095604496.
- 16 G. Genta G. Barbato, E. M. Barini and R. Levi. Features and performance of some outlier detection methods. *Journal of Applied Statistics*, 38(10):2133–2149, 2011. doi:10.1080/02664763.2010.545119.
- 17 James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. The java® language specification, 2024. URL: <https://docs.oracle.com/javase/specs/jls/se22/jls22.pdf>.
- 18 Tobias Hartmann, Albert Noll, and Thomas R. Gross. Efficient code management for dynamic multi-tiered compilation systems. In Joanna Kolodziej and Bruce R. Childers, editors, *2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14, Cracow, Poland, September 23-26, 2014, PPPJ '14*, pages 51–62, New York, NY, USA, 2014. ACM. doi:10.1145/2647508.2647513.
- 19 Urs Hözle, Craig Chambers, and David M. Ungar. Debugging optimized code with dynamic deoptimization. In Stuart I. Feldman and Richard L. Wexelblat, editors, *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI), San Francisco, California, USA, June 17-19, 1992*, volume 27, pages 32–43, New York, NY, USA, July 1992. ACM. doi:10.1145/143095.143114.
- 20 Kenneth Hoste and Lieven Eeckhout. Cole: compiler optimization level exploration. In Mary Lou Soffa and Evelyn Duesterwald, editors, *Sixth International Symposium on Code Generation and Optimization (CGO 2008), April 5-9, 2008, Boston, MA, USA, CGO '08*, pages 165–174, New York, NY, USA, 2008. ACM. doi:10.1145/1356058.1356080.
- 21 Christian Hummer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. A domain-specific language for building self-optimizing AST interpreters. In Ulrik Pagh Schultz and Matthew Flatt, editors, *Generative Programming: Concepts and Experiences, GPCE'14, Västerås, Sweden, September 15-16, 2014*, volume 50, pages 123–132, New York, NY, USA, September 2014. ACM. doi:10.1145/2658761.2658776.

- 22 Tarindu Jayatilaka, Hideto Ueno, Giorgis Georgakoudis, Eunjung Park, and Johannes Doerfert. Towards compile-time-reducing compiler optimization selection via machine learning. In Federico Silla and Osni Marques, editors, *ICPP Workshops 2021: 50th International Conference on Parallel Processing, Virtual Event / Lemont (near Chicago), IL, USA, August 9-12, 2021, ICPP Workshops '21*, pages 23:1–23:6, New York, NY, USA, 2021. ACM. doi:10.1145/3458744.3473355.
- 23 Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth B. Russell, and David Cox. Design of the java hotspot™ client compiler for java 6. *ACM Trans. Archit. Code Optim.*, 5(1):7:1–7:32, May 2008. doi:10.1145/1369396.1370017.
- 24 Sameer Kulkarni and John Cavazos. Mitigating the compiler optimization phase-ordering problem using machine learning. 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, OOPSLA '12*, pages 147–162, New York, NY, USA, 2012. ACM. doi:10.1145/2384616.2384628.
- 25 Octave Larose, Sophie Kaleba, Humphrey Burrell, and Stefan Marr. AST vs. bytecode: Interpreters in the age of meta-compilation. *Proc. ACM Program. Lang.*, 7(OOPSLA2):318–346, October 2023. doi:10.1145/3622808.
- 26 Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, and Daniel Smith. The java® virtual machine specification, 2024. URL: <https://docs.oracle.com/javase/specs/jvms/se22/jvms22.pdf>.
- 27 Stefan Marr, Benoit Daloz, and Hanspeter Mössenböck. Cross-language compiler benchmarking: are we fast yet? In Roberto Ierusalimschy, editor, *Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016, Amsterdam, The Netherlands, November 1, 2016*, volume 52, pages 120–131, New York, NY, USA, November 2016. ACM. doi:10.1145/2989225.2989232.
- 28 Uwe Meyer. Techniques for partial evaluation of imperative languages. 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, PEPM '91*, pages 94–105, New York, NY, USA, 1991. ACM. doi:10.1145/115865.115876.
- 29 Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
- 30 Gennady Pekhimenko and Angela Demke Brown. Efficient program compilation through machine learning techniques. In Ken Naono, Keita Teranishi, John Cavazos, and Reiji Suda, editors, *Software Automatic Tuning, From Concepts to State-of-the-Art Results*, pages 335–351. Springer, New York, NY, 2010. doi:10.1007/978-1-4419-6935-4_19.
- 31 Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, June 1975. doi:10.1145/360825.360839.
- 32 Dmitry Plotnikov, Dmitry Melnik, Mamikon Vardanyan, Ruben Buchatskiy, Roman Zhuykov, and Je-Hyung Lee. Automatic tuning of compiler optimizations and analysis of their impact. In Vassil Alexandrov, Michael Lees, Valeria V. Krzhizhanovskaya, Jack J. Dongarra, and Peter M. A. Sloot, editors, *Proceedings of the International Conference on Computational Science, ICCS 2013, Barcelona, Spain, 5-7 June, 2013*, volume 18 of *Procedia Computer Science*, pages 1312–1321. Elsevier, 2013. 2013 International Conference on Computational Science. doi:10.1016/j.procs.2013.05.298.
- 33 Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. Bringing low-level languages to the JVM: efficient execution of LLVM IR on truffle. In Antony L. Hosking and Witawas Srisa-an, editors, *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages, VMIL@SPLASH 2016, Amsterdam, The Netherlands, October 31, 2016, VMIL 2016*, pages 6–15, New York, NY, USA, 2016. ACM. doi:10.1145/2998415.2998416.

- 34 Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Icebreaker: warming serverless functions better with heterogeneity. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, ASPLOS '22, pages 753–767, New York, NY, USA, 2022. ACM. doi:[10.1145/3503222.3507750](https://doi.org/10.1145/3503222.3507750).
- 35 Doug Simon, Christian Wimmer, Bernhard Urban, Gilles Duboscq, Lukas Stadler, and Thomas Würthinger. Snippets: Taking the high road to a low level. *ACM Trans. Archit. Code Optim.*, 12(2):20:20:1–20:20:25, June 2015. doi:[10.1145/2764907](https://doi.org/10.1145/2764907).
- 36 Matija Sipek, Branko Mihaljević, and Aleksander Radovan. Exploring aspects of polyglot high-performance virtual machine graalvm. In Marko Koricic, Zeljko Butkovic, Karolj Skala, Zeljka Car, Marina Cicin-Sain, Snjezana Babic, Vlado Sruk, Dejan Skvorc, Slobodan Ribaric, Stjepan Gros, Boris Vrdoljak, Mladen Mauher, Edvard Tijan, Predrag Pale, Darko Huljenic, Tihana Galinac Grbac, and Matej Janjic, editors, *42nd International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2019, Opatija, Croatia, May 20-24, 2019*, pages 1671–1676. IEEE, 2019. doi:[10.23919/MIPRO.2019.8756917](https://doi.org/10.23919/MIPRO.2019.8756917).
- 37 Matija Sipek, D. Muhamagic, Branko Mihaljević, and Aleksander Radovan. Enhancing performance of cloud-based software applications with graalvm and quarkus. In Marko Koricic, Karolj Skala, Zeljka Car, Marina Cicin-Sain, Vlado Sruk, Dejan Skvorc, Slobodan Ribaric, Bojan Jerbic, Stjepan Gros, Boris Vrdoljak, Mladen Mauher, Edvard Tijan, Tihomir Katulic, Predrag Pale, Tihana Galinac Grbac, Nikola Filip Fijan, Adrian Boukalov, Dragan Cisic, and Vera Gradisnik, editors, *43rd International Convention on Information, Communication and Electronic Technology, MIPRO 2020, Opatija, Croatia, September 28 - October 2, 2020*, pages 1746–1751. IEEE, 2020. doi:[10.23919/MIPRO48935.2020.9245290](https://doi.org/10.23919/MIPRO48935.2020.9245290).
- 38 Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. Partial escape analysis and scalar replacement for java. In David R. Kaeli and Tipp Moseley, editors, *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, Orlando, FL, USA, February 15-19, 2014*, CGO '14, page 165, New York, NY, USA, 2014. ACM. doi:[10.1145/2581122.2544157](https://doi.org/10.1145/2581122.2544157).
- 39 James Stanier and Des Watson. Intermediate representations in imperative compilers: A survey. *ACM Comput. Surv.*, 45(3):26:1–26:27, July 2013. doi:[10.1145/2480741.2480743](https://doi.org/10.1145/2480741.2480743).
- 40 Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, April 1991. doi:[10.1145/103135.103136](https://doi.org/10.1145/103135.103136).
- 41 Christian Wimmer. Linear scan register allocation for the java hotspotTM client compiler, 2004.
- 42 Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. Initialize once, start fast: application initialization at build time. *Proc. ACM Program. Lang.*, 3(OOPSLA):184:1–184:29, October 2019. doi:[10.1145/3360610](https://doi.org/10.1145/3360610).
- 43 W. Eric Wong, Joseph R. Horgan, Saul London, and Hiralal Agrawal. A study of effective regression testing in practice. In *Eighth International Symposium on Software Reliability Engineering, ISSRE 1997, Albuquerque, NM, USA, November 2-5, 1997*, pages 264–274. IEEE Computer Society, November 1997. doi:[10.1109/ISSRE.1997.630875](https://doi.org/10.1109/ISSRE.1997.630875).
- 44 Thomas Würthinger, Christian Wimmer, Christian Hummer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, PLDI 2017, pages 662–676, New York, NY, USA, 2017. ACM. doi:[10.1145/3062341.3062381](https://doi.org/10.1145/3062341.3062381).
- 45 Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Hummer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to rule them all. In Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld, editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, Onward! 2013, pages 187–204, New York, NY, USA, 2013. ACM. doi:[10.1145/2509578.2509581](https://doi.org/10.1145/2509578.2509581).

- 46 Yifei Zhang, Tianxiao Gu, Xiaolin Zheng, Lei Yu, Wei Kuai, and Sanhong Li. Towards a serverless java runtime. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*, pages 1156–1160. IEEE, 2021. doi:10.1109/ASE51524.2021.9678709.

A Frame-State Algorithm

Listing 3 Pseudo-code representation of the updated deletion algorithm.

```

1   void deleteUnusedNodes(List<Node> deleteList) {
2       // track the usages of each node
3       Map<Node, Int> usages = new Map();
4       Set<Node> maybeDelete = new Set();
5
6       // delete the initial set of nodes
7       for (Node n in deleteList) {
8           delete(n);
9           for (Node input in n.inputs()) {
10              Int u = usages[input];
11              if (u == null) {
12                  u = input.usageCount();
13              }
14              usages[input] = u - 1;
15              maybeDelete.add(input);
16          }
17      }
18
19      // fixed point iteration to delete nodes transitively
20      for (Node n in maybeDelete) {
21          if (shouldBeDeleted(n, u)) {
22              delete(n);
23              for (Node input in n.inputs()) {
24                  Int u = usages[input];
25                  if (u == null) {
26                      u = input.usageCount();
27                  }
28                  usages[input] = u - 1;
29                  maybeDelete.add(input);
30              }
31          }
32      }
33
34      // remove the usages of nodes that were not deleted
35      // and for which the usage count changed
36      for (Node n, Int u in usages) {
37          if (isAlive(n) && n.usageCount() != u) {
38              n.removeDeadUsages();
39          }
40      }
41  }
```

Learning Gradual Typing Performance

Mohammad Wahiduzzaman Khan  

CACS, University of Louisiana, Lafayette, LA, USA

Sheng Chen  

CACS, University of Louisiana, Lafayette, LA, USA

Yi He  

Data Science, College William & Mary, Williamsburg, VA, USA

Abstract

Gradual typing has emerged as a promising typing discipline for reconciling static and dynamic typing, which have respective strengths and shortcomings. Thanks to its promises, gradual typing has gained tremendous momentum in both industry and academia. A main challenge in gradual typing is that, however, the performance of its programs can often be unpredictable, and adding or removing the type of a single parameter may lead to wild performance swings. Many approaches have been proposed to optimize gradual typing performance, but little work has been done to aid the understanding of the performance landscape of gradual typing and navigating the migration process (which adds type annotations to make programs more static) to avert performance slowdowns.

Motivated by this situation, this work develops a machine-learning-based approach to predict the performance of each possible way of adding type annotations to a program. On top of that, many supports for program migrations could be developed, such as finding the most performant neighbor of any given configuration. Our approach gauges runtime overheads of dynamic type checks inserted by gradual typing and uses that information to train a machine learning model, which is used to predict the running time of gradual programs. We have evaluated our approach on 12 Python benchmarks for both guarded and transient semantics. For guarded semantics, our evaluation results indicate that with only 40 training instances generated from each benchmark, the predicted times for all other instances differ on average by 4% from the measured times. For transient semantics, the time difference ratio is higher but the time difference is often within 0.1 seconds.

2012 ACM Subject Classification Theory of computation → Type structures; Computing methodologies → Machine learning; Computing methodologies → Learning linear models

Keywords and phrases Gradual typing performance, type migration, performance prediction, machine learning

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.21

Supplementary Material Software (*Source Code*): https://github.com/wahid-nlogn/ECOOP_2024_MLGP, archived at [swh:1:dir:3f8fb3b4e2160fa825b6f823185bf8e2a7e1ec92](https://doi.org/10.4230/wh:1:dir:3f8fb3b4e2160fa825b6f823185bf8e2a7e1ec92)

Funding This work has been supported in part by the National Science Foundation (NSF) under Grant Nos. IIS-2245946, IIS-2236578, and CCF-1750886 and in part by the Commonwealth Cyber Initiative (CCI) and DARPA.

1 Introduction

Statically typed languages offer benefits such as early programming error detection, documentation, and better performance but can hinder program executions when they are incomplete or contain type errors. Dynamically-typed languages offer the benefits of fast prototyping and flexible usability but provide less program correctness guarantee. Traditionally, languages are either static or dynamic. In an effort to reconcile these typing disciplines, a typing discipline named *gradual typing* was developed and popularized in the last decade Siek and Taha [38],

21:2 Learning Gradual Typing Performance

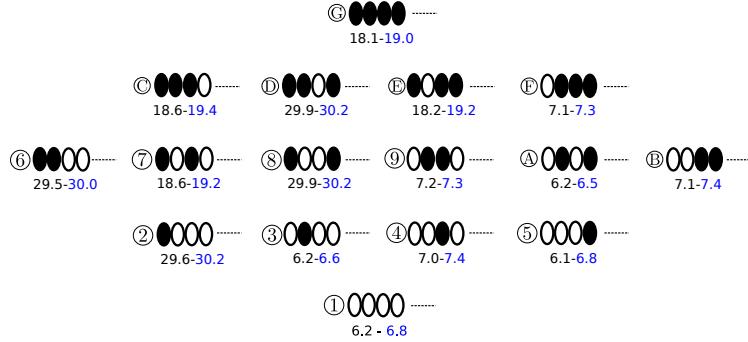


Figure 1 Part of the performance lattice for the Pascal benchmark. The lattice consists of 16 configurations, a combination of four parameters with each being typed or untyped. Each filled (unfilled) oval represents a typed (untyped) parameter. Each configuration shows only 4 ovals and omits the rest, which is the same across the 16 configurations. A circled number or letter is attached to each configuration for easy reference in the paper. Each configuration is associated with two times, separated by a ‘-’. The first time is the measured time of the configuration and the second (in blue) is the predicted time by our machine learning algorithm. All times are in seconds in the paper.

Siek and Vachharajani [39], Garcia and Cimini [13], Tobin-Hochstadt and Felleisen [42], Tobin-Hochstadt et al. [43], Campora et al. [6], Castagna et al. [9], Migeed and Palsberg [22], Phipps-Costin et al. [31], Greenman and Felleisen [14].

The main idea of gradual typing is that within a single program, parts of it may be statically typed (by giving type annotations to parameters in that part) and parts of it may be dynamically typed (by leaving out type annotations to parameters or explicitly giving them the dynamic type, written as `dyn`). Ideally, in gradual typing, prototyping and initial development is done with the dynamic aspect of the language, and programs are migrated to static aspect when performance and correctness becomes critical.

The goal of type migration is to add type annotations to parameters with dynamic types of a program. A commonly used notion in type migration is *configurations* Greenman et al. [15]. For any program, a configuration specifies which subset of all the parameters are typed. For example, in the fully dynamic configuration, this subset is empty, and in the fully static configuration, this subset includes all parameters. For a program with n parameters, there can be up to 2^n configurations since each parameter can be typed or untyped. We can organize all the configurations into a lattice such that the set of typed parameters in the join of two configurations is a union of those of the two configurations. To illustrate, Figure 1 presents a part of the lattice for the Pascal benchmark in Python.

1.1 Performance Problem in Type Migration

There are two issues related to gradual type migration:

- (1) finding parameters where type annotations could be added and
- (2) understanding performance changes and maintaining good (acceptable) performance as type annotations are added.

For issue 1, a lot of work has been done to automatically adding type annotations to dynamically-typed programs, including static approaches Castagna et al. [9], Kristensen and Møller [19], Campora and Chen [7], Rastogi et al. [34], Chandra et al. [10], Siek and Vachharajani [39], dynamic approaches Miyazaki et al. [24], Cristiani and Thiemann [11], and machine learning based approaches Mir et al. [23], Peng et al. [30], Pradel et al. [33],

Allamanis et al. [3]. Several approaches have also been developed to find best migrations in the sense of adding type annotations to as many parameters as possible Campora et al. [6], Migeed and Palsberg [22], Phipps-Costin et al. [31]. Issue 2, however, has received less attention.

While it is tempting to integrate all the type annotations suggested by a type migration tool Castagna et al. [9], Kristensen and Møller [19], Campora and Chen [7], Rastogi et al. [34], Chandra et al. [10], Siek and Vachharajani [39], Mir et al. [23], Peng et al. [30], Pradel et al. [33], Allamanis et al. [3], doing so may turn the original configuration into a new one that degrades performance significantly. The slowdown can be as high as more than 100 times Takikawa et al. [41], due to intricate type interactions. This is the case even when the type annotations for all the parameters in a single project are inferred. For example, in the spectral norm benchmark, the runtime for the fully typed configuration is about 2 times that of a configuration that has one fewer function typed Campora et al. [8]. The reason is that even all parameters in a project are typed, the libraries and third-party code used by the project may not be typed.

In general, after migrating from configuration K_s to K_e , manually or with the aid of type migration tools, the developer may face a few performance related questions. In particular, if the performance at K_e is not satisfactory, then the user will have to explore the performance of the neighbors of K_e to find a configuration that can restore the performance at K_s or whose performance is the best among all neighbors.

To illustrate, consider the performance lattice for the Pascal benchmark in Figure 1. The Pascal program has 19 parameters and thus 2^{19} configurations, and we present a part of the lattice in the figure. Assume the user is currently at configuration ① and a migration tool infers types for the four parameters, which corresponds to configuration ⑥. However, noting that the performance at ⑥ is about 3 times slower than that at ①, the user will explore the performance of neighbors and find one with good performance.

The problem is that there is no obvious strategy Greenman et al. [15], Takikawa et al. [41] that the user could employ to quickly find desired configurations. For example, a strategy like breadth-first-search will not find ⑥, the configuration that both has good performance and has largest number of parameters being typed, without trying ②, ③, and ④. Similarly, a strategy like depth-first-search will not find any configuration that restores performance until it goes back to the original configuration ①. This problem will become worse in practice due to three reasons. First, type migration tools may suggest adding types to many more parameters, which quickly enlarges the search space. Second, as the program becomes bigger, it takes more time to measure the performance of each configuration. Also, it takes more time to move from one configuration to another as more type changes are involved. Third, since each program has its own structure and type of interactions, as witnessed by very diverse performance lattices in different programs Takikawa et al. [41], Greenman et al. [16], Campora et al. [8], no single searching strategy works well for all programs.

The biggest problem is probably the uncertainty associated with the exploration process. If the user has not found a configuration with good performance following some strategy, should the user stick to the strategy in hoping that the performance will finally improve or change the strategy in fearing that performant configurations are in other neighborhoods.

1.2 A Machine Learning Based Solution

In this paper, we propose and develop `LearnPerf`, a machine learning based solution for this problem. For each program, we train a model from the running time for a very limited number (usually 40) of configurations. We then use this model to predict the execution times of other configurations. To give a sense of how the predicted times of `LearnPerf` look like, we present them in Figure 1 in blue.

Our prediction result is pretty accurate, with the *difference ratio* (defined as $|\text{predicted time} - \text{measured time}|/\text{measured time}$) often within 4%. On top of the prediction result, we can develop a series of migration support under different scenarios. We list some of them below.

1. `LearnPerf` is able to predict the performance for a given configuration. Assume the developer wants to migrate the current configuration to a new one, this information can inform how performance looks like at the new configuration.
2. `LearnPerf` is able to classify the performance of adjacent configurations. For a certain number of configurations around the current one, we can classify them according to performance speedup/slowdown scales. Takikawa et al. [41] introduced the notion 2-deliverable, which includes all configurations whose performance degrades by less than 2 times that of the original configuration, and 2-5 usable, which includes all configurations that slows down the original configuration by 2-5 times. With the help of `LearnPerf`, we can highlight configurations that are 2-deliverable, 2-5 usable, etc.
3. For each configuration, `LearnPerf` is able to find the most performant configurations within its neighborhood. If the user is not satisfied with the performance of the current configuration, this capability can suggest an alternating configuration with good performance.

Note that this work studies the performance aspect of type migration only and is not intended to develop a new type inference algorithm or machine learning algorithm to automatically add type annotations. As mentioned earlier, there has been a long line of research of adding type annotations but little work has been done for the performance aspect except for two papers Campora et al. [8], Greenman et al. [15]. Many approaches Feltey et al. [12], Ortin et al. [28], Moy et al. [25], Kuhlenschmidt et al. [20], Vitousek et al. [46] have investigated performance optimization of gradual typing. Our work is orthogonal to these approaches and we discuss the relation with them in Section 6.

To illustrate the usefulness of our migration support, assume the user was at configuration ① and has just migrated to ⑥ and observed that the performance at ⑥ is not satisfactory. `LearnPerf` can help in this case. For example, there are four neighbors of ⑥, ⑦ through ⑩. `LearnPerf` predicts that their running times are 19.4, 30.2, 19.2, and 7.3 seconds, respectively. Based on the predicted times, `LearnPerf` suggests the user to migrate to ⑩, rather than ⑥. We can observe that ⑩ slows down ① by only 15% while ⑥ slows down by 3 times.

Overall, `LearnPerf` finds ⑩ that reconciles both performance and static migration. It seems undesirable to migrate to ⑩ and not ⑥ because ⑥ adds a type annotation to one more parameter. In practice, this problem can often be solved by migrating a few parameters in unison in later migrations.

1.3 Workflow and Contributions of This Work

Figure 2 presents the workflow of `LearnPerf`. Starting from any “original program”, assume the user added some type annotations, manually or with the help of type migration tools. This will lead to a new, more precise program that has more type annotations than the original program. Note, that the new program may be partially typed or fully typed. As discussed earlier, both partial and fully typed programs may experience significant performance degradation. From the new program, we create 40 random training samples. We then extract relevant feature vectors that characterizes runtime performance as well as the running time for each generated configuration. This information is then input into `LearnPerf` for the purpose of training our model.

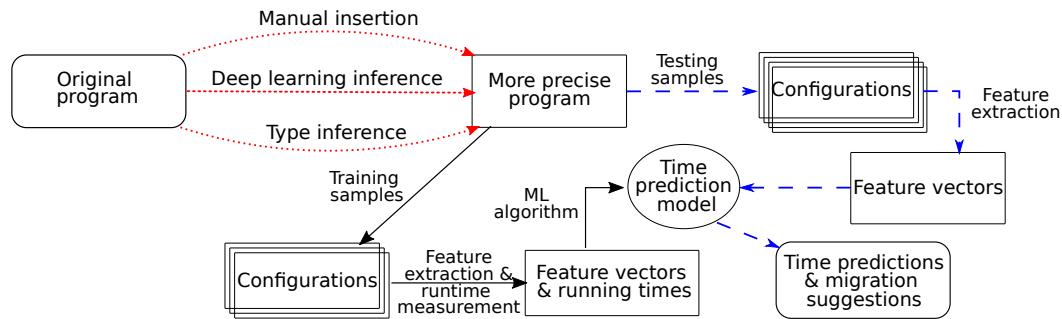


Figure 2 Workflow of LearnPerf. Solid arrows denote information flow in training phase and dashed, blue arrows denote that in prediction phase. Dotted, red arrows denote type annotation additions, and they are not a part of this work.

Once we have the trained model, we can support scenarios 1 through 3 by generating appropriate configurations and predicting their running times. Section 5 will sketch the main steps to support scenarios 2 and present our evaluation results.

In the above, the training will not start until the user initiates it. In practice, however, our approach will actively invoke a deep learning model Mir et al. [23], Peng et al. [30], Pradel et al. [33], Allamanis et al. [3] to generate type annotations. As a result, the performance prediction model could be ready before the programmer starts the migration process and needs migration support. We applied this idea to three large datasets for our evaluation (Section 5).

Overall, this paper makes the following contributions:

1. We develop a machine-learning based approach that can help understand the performance landscape of different configurations of a gradually-typed program. On top of that, many migration supports can be developed.
2. We explore different features to represent program run times and find out that overheads of casts inserted by gradual typing are simple yet representative features.
3. We implement our approach and evaluate its performance on twelve benchmarks, including nine benchmarks that are frequently used in gradual typing research and three larger benchmarks that each have more than 1000 LOC. We observe that with only 40 training instances, our predicted times differ from measured times by 4% only for guarded semantics. For transient semantics, the difference between the predicted time and measured time is often within 0.1 seconds.

The rest of the paper is organized as follows. In Section 2, we discuss the background of gradual typing. In Sections 3 and 4, we present our exploration of searching for appropriate machine learning model and representative features for precisely estimating execution times of gradual programs. In Section 5, we present the evaluation results, as well as implementation details and benchmarks used. We discuss related work in Section 6 and conclude in Section 7.

2 Background

This section covers the background of gradual typing, with a focus on cast insertions and their overheads.

In gradual typing, a parameter may be given a static type, a dynamic type (often written as `Dyn` or is omitted) signifying that the type is not known statically, or a mix of static and dynamic types. Static type checking is applied to program parts that use parameters with static types, and dynamic type checking is used for other program parts.

21:6 Learning Gradual Typing Performance

```

def myreduce(f, lst, init):
    result = init
    for i in range(len(lst)):
        result = f(result, lst[i])
    return result

def wider(cw:Int, ci>List(Int))>Int:
    return max(cw, len(ci))

myreduce(wider, [[1], [], [4,5]], 0)

```



```

def myreduce(f, lst, init):
    result = init
    for i in range(len(lst)):
        result = (f : Dyn => Dyn -> Dyn -> Dyn)(result, lst[i])
    return result

def wider(cw, ci):
    return max(cw, len(ci)) : Dyn => Int

myreduce(wider: Int -> List(Int) -> Int => Dyn,
        [[1], [], [4,5]] : List(List(Int)) => Dyn,
        0 : Int => Dyn)

```

 **Figure 3** A partially-typed version of `myreduce` (left) and its cast-inserted program (right).

For example, Figure 3 (left) presents a program snippet written in a hypothetical gradual language in Python type hint syntax Vitousek et al. [45]. The function `myreduce` takes in a binary function, a list, and an initial value and reduces the list to a single value. In this program, static type annotations are given to the parameters, and the return of `wider`. All other parameters have dynamic types. A static type error will be detected if we pass a string value as the first argument to `wider` because the first parameter has a type annotation `Int`. In contrast, no such error will be detected if we pass a string value as the first argument to `myreduce`.

Gradually-typed languages are often obtained by adding static type checking to underlie dynamic languages, such as Typed Clojure for Clojure, Typed Racket for Racket, and Reticulated Python Vitousek et al. [45] for Python. As such, a common implementation strategy of gradual typing is to translate its programs into programs in the underlying language and insert necessary runtime type checks (often called casts) during the translation.

For example, when executed by a gradual typing implementation for Python, the program in Figure 3 (left) is translated to the program in Figure 3 (right), which can be executed on any Python interpreter. Comparing programs in Figure 3 left and right, we observe two important differences. First, the program on the right does not have type annotations. This is because the interpreter for the underlying, dynamic language often does not make use of type annotations so they are erased during translation. Second, the program on the right has extra constructs in the form of `expr : src_type => trg_type`, which are often called casts. Such casts are inserted when the static type checker determines that `expr` has the type `src_type` but is used in a context where a value of `trg_type` is required.

As runtime type checks, these casts incur runtime overheads, and different casts lead to very different overheads. For example, the cast `x : Dyn => Int` can be performed where it appears as we can always verify if `x` is indeed an integer and is thus very lightweight. In contrast, the cast `g : Dyn => Int -> Bool` can not be verified where it appears because, for example, we do not know how `g` will be used and what arguments will be passed to it. As such, a proxy will be created for `g` such that the invocation of `g` will be handled by the proxy, which inserts a cast to check that the argument to `g` is `Int` and another cast to check that the return value of `g` is `Bool`. Such casts are more involved and lead to more significant overheads. Casts over data structures and objects are similarly heavyweight.

It is not hard to envision that adding or removing the type annotation for a single parameter in gradual typing may yield significant performance swings Takikawa et al. [41], Greenman et al. [16]. One might consider this a reason to abandon gradual language designs that enforce type invariants at runtime, but a study by Tunnell Wilson et al. [44] shows that programmers often expected the behavior of programs to emulate those done by gradual typing. This work in this paper enables programmers to enjoy the benefits of gradual typing while staying informed about the performance landscape as they migrate programs toward more static.

■ **Table 1** Deep learning model performance on unseen benchmarks.

Unseen Benchmark	# training	# testing	MAE	MSE	<i>DR</i>
Meteor	105945	1024	5.26	28.09	56.84%
Zebrafy	103969	3000	149.37	22350.18	86.96%
Pascal	101785	5184	22.82	653.85	273.42%
Chaos	100969	6000	25.67	784.96	35.57%
Richard	100969	6000	23.73	766.28	36.05%
Sieve	91608	15361	53.88	2933.99	1655.90%
Nbody	90585	16384	4.87	36.79	68.28%
Scimark	81881	25088	4.39	28.27	80.37%
Raytrace	73065	33904	6.49	46.29	110.17%

3 Feature Engineering

The two most important questions in machine learning are what kinds of models to train and what features will be used for representing programs. In this section and next, we present our exploration of searching for a suitable model and simple yet representative features.

3.1 First Attempt: Global Model with Deep Learning

Ideally, we train a *global* model that can be used to predict the runtime of different configurations for all user programs. Such a model needs to be trained only once by us (the model developer) and can be distributed to users (developers who migrate gradual programs) for use.

Motivated by recent successes of deep learning models for predicting types Mir et al. [23], Peng et al. [30], Pradel et al. [33], Allamanis et al. [3], our first attempt is to exploit deep learning to train a global model. For a given set of configurations for training and a set for testing, this process consists of several steps. The main challenge here is that the training instances may have different lengths. To solve this issue, we leverage source code embeddings that convert each configuration into an embedding that has the same length. Specifically, we use UniXcoder Guo et al. [17] to convert each configuration into a 4 * 768 float matrix. These embeddings, together with runtimes of corresponding configurations, are fed into a multi-layer perception network Popescu et al. [32] to train a global model. Based on the trained model, we can predict the runtime for each configuration in the test set.

To test the performance of this idea, we have developed a prototype and conducted experiments in two settings. In the first setting, we collected all configurations from nine benchmarks (listed in Table 1), with a total of 106,969 configurations (Section 5.1 will give more details about our evaluation benchmarks). We randomly choose 80% of them for training, 10% for cross-validation, and 10% for testing. In the second setting, we chose one benchmark for testing and used configurations from all other benchmarks for training. The main difference between these two settings is that in the first setting some testing configurations and training configurations may come from the same benchmark.

To measure the performance of this exploration and later ones in this paper, we use two of the most popular metrics for a regression problem, mean absolute error (*MAE*) and mean square error (*MSE*). In addition, to capture the accuracy or error ratio more intuitively, we used another metric called difference ratio, shortened to *DR*. The definitions of these three metrics are given below, where t_i and \hat{t}_i denote the measured and predicted running times of

the configuration i , respectively, and D denotes the testing set of instances. For example, if the measured and predicted times for a configuration is 7.9s and 8.1s, respectively, then the difference ratio for this configuration is 2.53%. We will use these notations throughout the paper.

$$MAE = \frac{\sum_{i=1}^D |t_i - \hat{t}_i|}{|D|} \quad MSE = \frac{\sum_{i=1}^D (t_i - \hat{t}_i)^2}{|D|} \quad DR = \frac{\sum_{i \in D} \frac{|t_i - \hat{t}_i|}{t_i}}{|D|}$$

The DR for the first setting is 147.58%. The details of the results for the second setting is given in Table 1. The results show that the global model trained with deep learning performs poorly. There are a few possible reasons. First, as discussed in Section 2, a gradual program is often translated to a base program in the untyped, underlying language with casts inserted. As such, the running time of a configuration roughly includes the time to execute the base program and the overhead due to casts. To be able to precisely predict the running time, we need to be able to do that for both parts. However, predicting the running time of a general program is still an open problem Matsunaga and Fortes [21]. Second, the overhead due to casts can vary significantly across different programs as it depends on program structures, such as whether casts are in loops, whether multiple casts are applied to single values, etc. Third, as discussed in Section 2, two configurations that differ by whether a single parameter is typed or not may have very different runtimes. This exhibits similar phenomena as in molecular property prediction where minor changes in molecular structures lead to significant changes of properties Stumpfe and Bajorath [40]. Earlier work Xia et al. [50] has demonstrated that deep models often do not perform well for such tasks.

For this reason, we decide to train an individual, project-specific model for each project in this work. Having decided which model to train, we next explore different feature representations to find representative features.

3.2 Second Attempt: Individual Models with Bit Strings

The problem of predicting gradual typing performance bears some similarity to performance prediction for highly-configurable software systems Kolesnikov et al. [18]. A highly-configurable program usually contains a large number of configuration options (for example, Linux has about 13,000 such options) for customizing the functional and non-functional features of the program. For instance, Linux can be configured to run on a diverse set of devices, ranging from embedding devices to servers. Each configuration option may be set or unset, corresponding to enabling or disabling associated features, which often leads to the inclusion or exclusion of certain pieces of code into the generated program after customization. As such, different configurations of the same configurable program will lead to different performances.

Understanding the performance landscape of configurable software systems is an important research problem, particularly as generating all possible programs and measuring their performance is infeasible due to the exponential complexity (the number of different configurations that can be generated is exponential in the number of configuration options). A prevalent solution to this problem is building a *performance-influence* model for each configurable software system. This can be achieved by generating a few samples, measuring the performance of these samples, and building a model from them. With the *performance-influence* model, predicting the performance of a certain configuration is instantaneous, without having to generate the configuration and measure the performance.

■ **Table 2** Python benchmark Performance (Bit strings).

Benchmark	# training	# testing	MAE	MSE	<i>DR</i>
Monte Carlo	40	344	0.53 ± 0.00	0.45 ± 0.00	35.30%
Meteor	40	984	0.29 ± 0.02	0.19 ± 0.068	2.47%
CPU	40	2857	2.57 ± 0.06	3.487 ± 0.08	8.15%
Zebrafy	40	3960	12.25 ± 1.24	15.98 ± 0.78	91.64%
Pascal	40	5144	5.15 ± 0.18	6.44 ± 0.31	27.57%
Chaos	40	5960	2.38 ± 0.04	3.00 ± 0.06	4.78%
Richard	40	5960	9.58 ± 1.12	13.65 ± 1.68	42.88%
BenchFirst	40	5960	43.85 ± 3.88	58.58 ± 4.23	25.88%
Sieve	40	15321	0.12 ± 0.00	0.16 ± 0.00	1.56%
Nbody	40	16344	3.45 ± 0.17	4.46 ± 0.19	30.10%
Scimark	40	25048	2.41 ± 0.03	3.06 ± 0.05	17.02%
Raytrace	40	33864	5.80 ± 1.23	7.35 ± 1.60	38.90%
Monte Carlo	192	192	0.39 ± 0.00	0.47 ± 0.00	31.33%
Meteor	512	512	0.13 ± 0.00	0.08 ± 0.00	0.70%
CPU Benchmark	1427	1428	2.23 ± 0.03	2.84 ± 0.02	6.55%
Zebrafy	2000	2000	10.72 ± 0.95	14.58 ± 0.89	87.24%
Pascal	2592	2592	3.81 ± 0.00	5.09 ± 0.00	20.45%
Chaos	3000	3000	1.72 ± 0.01	2.13 ± 0.00	3.41%
Richard	3000	3000	8.86 ± 0.01	13.05 ± 0.02	36.78%
BenchFirst	3000	3000	28.85 ± 3.05	36.58 ± 3.90	11.88%
Sieve	7680	7680	0.10 ± 0.00	0.13 ± 0.00	1.32%
Nbody	8192	8192	2.70 ± 0.00	3.58 ± 0.00	23.64%
Scimark	12544	12544	1.75 ± 0.00	2.42 ± 0.00	12.36%
Raytrace	16952	16952	2.81 ± 0.00	3.19 ± 0.00	18.84%

In gradual typing, each parameter can be typed or untyped, corresponding to enabling or disabling a configuration option. Due to this similarity, we started our exploration by using bit-string as features for machine learning. Specifically, we treat each parameter as a binary feature and use 1 to denote that the parameter is typed and 0 to denote it is untyped. Feature values for all parameters are concatenated together to form a bit-string, which forms the feature vector in this exploration.

We developed a prototype implementing this idea and tested its performance on 12 Python benchmarks (we will show details about them in Section 5.1). We present the result in Table 2. In the upper part of Table 2, we present the results with bit-strings as features when each individual model is trained with 40 configurations. We can observe that the average difference ratio (*DR*) is quite high for several benchmarks. For example, *DR* is around 92% for Zebrafy and 43% for Richard. We may think of increasing the number of training instances to boost the performance. Surprisingly, the performance does not increase significantly as we remarkably increase the number of training instances, as can be seen from the bottom part of Table 2. For example, as we increased the training instances from 40 to 2000 (that is we used 50% of instances for training) for Zebrafy, the average *DR* is still around 87%. Similarly, the average *DR* is about 37% for Nbody as we use 50% of all instances for training.

Another issue is that as we are training an individual model for each project, using too many training instances needs a very long preparation time. To solve this issue, we choose to generate a limited amount of configurations but extract highly effective features.

21:10 Learning Gradual Typing Performance

```

def myreduce(f:Function([Int,List(Int)],Int),
            lst>List(List(Int)), init:Int):
    result = init
    for i in range(len(lst)):
        result = f(result,lst[i])
    return result

def wider(cw:Int, ci>List(Int)) -> Int:
    return max(cw, len(ci))

myreduce(wider, [[1], [], [4,5]], 0)

```

```

def myreduce(f, lst, init):
    result = init
    for i in range(len(lst)):
        result = f(result : Dyn => Int,
                  lst[i]) : Int => Dyn
    return result

def wider(cw, ci):
    return max(cw, len(ci)) : Dyn => Int

myreduce(wider, [[1], [], [4,5]], 0)

```

Figure 4 The fully-typed version of `myreduce` (left) and its cast-inserted translation (right).

4 Third and Successful Attempt: Gauging Cast Overheads

The main reason that bit strings do not work well is that bits only represent whether parameters are typed or not while the types of parameters interact in an intricate way. This makes bit strings a poor candidate for capturing inserted casts, which are the main causes for performance overheads. For example, if we compare the programs in Figures 3 and 4, we can observe that while the program in Figure 4 (left) has strictly more type annotations than that in Figure 3, no such relation appears for the casts in the translated programs. In particular, these programs share only one common cast (the cast for the return value in `wider`), and all other casts are different. The running times of these two versions of `myreduce` are very different: the running time of the partially-typed version (Figure 3) is about 16 times that of the fully-typed version (Figure 4). In practice, removing or adding the type for a single parameter may lead to a completely different cast being inserted.

Thus, instead of using bit strings, we will next explore the inserted casts of the translated programs by gauging cast overheads. Our main idea is to give symbolic overheads to casts and let machine learning algorithm figure out the real overhead of each cast. To give a more formal account of our approach, we present the type syntax used for the rest of this section below.

$$\begin{array}{ll} \text{Base types} & U ::= \text{Bool} \mid \text{Int} \mid \text{Unit} \\ \text{Gradual types} & G ::= U \mid G \rightarrow G \mid \text{Dyn} \mid [G] \end{array}$$

Our type definition includes base types, ranged over by U , and gradual types, ranged over by G . Our base types include `Int`, `Bool`, and `Unit`, but they can be extended easily. In gradual types, we consider two type constructors: function types and list types. Again, they can be extended easily.

In the rest of this section, we first discuss how to gauge the overhead for individual casts (Sections 4.1 and 4.2) and then the overhead for a whole program (Section 4.3). Finally, we assess the effectiveness of cast overheads (Section 4.4).

4.1 Overheads for Individual Casts

Casts involving base types. Our first observation of gauging cast overheads is that casts have very different runtime overheads, as we discussed in Section 2. We first deal with casts that involve base types. For a cast of the form $U \Rightarrow \text{Dyn}$, it can be checked where it appears. We assign the symbolic overhead U^i to it. Similarly, for the cast $\text{Dyn} \Rightarrow U$, we assign the symbolic overhead U^p .

Casts involving function types. Next, we investigate overheads of casts that involve function types. In general, as discussed in Section 2, a function cast can not be verified where it appears. Instead, for a cast of the form $f: G_1 \rightarrow G_2 \Rightarrow G_3 \rightarrow G_4$, a proxy will be created to

wrap f . In place where f is called, the call is handled by the wrapper, which first casts the argument from G_3 to G_1 , calls f with the cast argument, and casts the return value of f from G_2 to G_4 . As such, a function cast induces two kinds of overheads: (1) the overhead that creates the proxy and (2) the overhead that casts the arguments and returned values. We refer to these two kinds of overheads as *creation overhead* and *invocation overhead*, respectively. The creation overhead should be similar across different proxy wrappers because type differences in casts do not cause the creation behavior to change much. As such, we assign F^c to represent a proxy creation overhead.

One challenge with invocation overheads is that they are incurred when the cast functions are invoked, not where the function casts appear. However, it is unclear when cast functions are invoked by looking at the translated program (neither with some standard static analysis) because cast functions may be assigned to other variables, stored in data structures, and passed over to other functions, and call sites can be very distant from where proxies are created. Our solution to this problem is to gauge the invocation overhead for each cast and directly add it to its creation overhead. This is very simple to implement: no complex alias analysis is needed.

Interestingly, this approach works well for predicting runtimes of configurations. Intuitively, the function cast created at the same program location will have the same invocation sites across different configurations since two configurations only differ by type annotations. Thus, if two casts cast the same function and have the same invocation overhead across two configurations, then they induce the same cast overheads. Of course, if the arguments to the cast functions in different configurations are cast differently, then the invocation takes different times to complete. However, such differences should be reflected through overhead differences of casts on the argument. Similarly, if the function cast in the first configuration has larger invocation overhead than that in the second configuration, then the cast function in the first configuration has more runtime overheads at invocation sites. We leave it to the machine learning algorithm that we use to train our model to figure out the relation between symbolic difference and the runtime difference for different configurations.

Another challenge in gauging invocation overheads is that unlike creation overheads that are similar across different function casts, invocation overheads can vary significantly, based on the types involved. For example, the cast $f1 : \text{Int} \rightarrow \text{Int} \Rightarrow \text{Dyn} \rightarrow \text{Dyn}$ should have a much smaller invocation overhead than $f2 : [\text{Int}] \rightarrow \text{Int} \Rightarrow \text{Dyn} \rightarrow \text{Dyn}$ because the cast for the argument for $f1$ is $\text{Dyn} \Rightarrow \text{Int}$ and that for $f2$ is $\text{Dyn} \Rightarrow [\text{Int}]$. As we have seen earlier, the cast $\text{Dyn} \Rightarrow \text{Int}$ is very lightweight while the cast $\text{Dyn} \Rightarrow [\text{Int}]$ involves the creation of another proxy over the argument (We will discuss casts involve lists later in this subsection), which will be treated as a list. Therefore, a plausible idea to accurately gauge invocation overheads is to assign different symbols for denoting different invocation overheads to different casts, based on their argument types and return types. The problem with this idea is that, however, we need to introduce a lot of different symbols for invocation overheads because within a program we could have many casts involving function types with different arities and different argument and return types. As we wanted to train our model with as few instances as possible, having too many symbols will negatively affect machine learning performance.

Our solution to this challenge is to break invocation overheads down and represent them with symbols we have already introduced. Our main insight is that an invocation overhead is originated from creating further casts at runtime. Thus, an invocation overhead can be approximated as a sum of all the creation overheads of the argument types and the return type. For example, for $f3 : (\text{Int} \rightarrow \text{Bool}) \rightarrow \text{Dyn} \Rightarrow \text{Dyn} \rightarrow \text{Int}$, the invocation overhead is creating a new function proxy for the argument to $f3$, which we have already introduced a symbol F^c , and

another cast for the $\text{Dyn} \Rightarrow \text{Int}$, which we used U^p to represent the overhead. Since the created function cast for the argument also introduces invocation overhead, we recursively apply this idea to the argument cast $\text{Dyn} \Rightarrow \text{Int} \rightarrow \text{Bool}$ and calculate its invocation overhead as $U^i + U^p$. Overall, the invocation overhead for the function cast $f3 : (\text{Int} \rightarrow \text{Bool}) \rightarrow \text{Dyn} \Rightarrow \text{Dyn} \rightarrow \text{Int}$ is $F^c + 2 \cdot U^p + U^i$. We give an algorithm for calculating cast overheads in Figure 5.

Casts involving list types. A cast involving list types, such as $l : \text{Dyn} \Rightarrow [\text{Bool}]$, also can not be verified where it appears because this cast ensures that future write accesses to l should add elements of type Bool only and future read accesses should get elements of Bool type. As such, similar to casts on function types, a proxy will be created for l and the proxy will make sure accesses to l have expected types. Therefore, the overhead of a list cast includes the creation overhead and access overhead. For the creation overhead, we use the symbol L^c to denote it.

For gauging access overheads, we face a challenge of locating where lists are accessed in the program, some of what we had for gauging invocation overheads for function casts. We adapt the solution there by gauging access overheads and add them to list creation overheads. For gauging access overheads themselves, the main insight is that list accessing can often be reduced to function calls Siek et al. [37], Vitousek et al. [45]. For example, for a list of type $[\text{Bool}]$, the function for ensuring that the element read from the list is Bool has the type $\text{Int} \rightarrow \text{Bool}$, where Int is the type of the parameter (list index) and Bool is the return type. The function for ensuring that the element added to the list is Bool has the type $\text{Int} \rightarrow \text{Bool} \rightarrow \text{Unit}$, where Int is the index type, Bool is the type of the element to be added to the list, and Unit is the return type of the function.

Based on this idea, the read access to the list l with the cast $l : \text{Dyn} \Rightarrow [\text{Bool}]$ can be reduced to the function cast $\text{Int} \rightarrow \text{Dyn} \Rightarrow \text{Int} \rightarrow \text{Bool}$, and the write access can be reduced to the cast $\text{Int} \rightarrow \text{Dyn} \rightarrow \text{Unit} \Rightarrow \text{Int} \rightarrow \text{Bool} \rightarrow \text{Unit}$. Thus, the access cost is approximated to be the cost of these two function casts. In practice, other operations may be performed on a list, such as insertion, extension, popping, and obtaining the length. However, read and write accesses are good representatives of access overheads because they are used frequently while others may not need function casts. Moreover, as we did in gauging invocation overheads, we only need to figure out the symbolic difference of list casts for the same list across different configurations, and let the machine learning algorithm scale that difference to appropriate runtime differences.

4.2 An Algorithm for Gauging Individual Casts' Overheads

We present an algorithm for gauging cast overheads in Figure 5. The algorithm is more general than our description in Section 4. For example, the algorithm deals with function casts that have multiple parameters. The algorithm is defined using the idea of pattern matching, and we assume that the most specific matching rule is used to handle the computation.

The main entry of the algorithm is the function $overHd$, which consists of eight cases. In the first case, the two types being cast are the same. Standard gradual typing implementations simply drop such casts, and so we assign 0 as its overhead. Cases two and three deal with casts between Dyn and function types, and we extend Dyn into a function type with the same arity as the function on the other side and delegate the computation to case eight of $overHd$. Cases four and five deal with casts between two function types that have different number of parameters. We assume that corresponding parameter types (such as G_1 and G'_1) and return types are consistent Siek and Taha [38]. We extend the type with fewer parameter types by padding it with DyNs . Cases six and seven deal with casts between Dyn and list types and are

$$\begin{aligned}
& \text{overHd}(G \Rightarrow G) = 0 \\
& \text{overHd}(\text{dyn} \Rightarrow G_1 \rightarrow \dots \rightarrow G_r) = \text{overHd}(\text{dyn} \rightarrow \dots \rightarrow \text{dyn} \Rightarrow G_1 \rightarrow \dots \rightarrow G_r) \\
& \text{overHd}(G_1 \rightarrow \dots \rightarrow G_r \Rightarrow \text{dyn}) = \text{overHd}(G_1 \rightarrow \dots \rightarrow G_r \Rightarrow \text{dyn} \rightarrow \dots \rightarrow \text{dyn}) \\
& \text{overHd}(G_1 \rightarrow \dots \rightarrow G_i \rightarrow \text{dyn} \Rightarrow G'_1 \rightarrow \dots \rightarrow G'_{i+j} \rightarrow G'_r) \\
& \quad = \text{overHd}(G_1 \rightarrow \dots \rightarrow G_i \rightarrow \text{dyn} \rightarrow \dots \rightarrow \text{dyn} \Rightarrow G'_1 \rightarrow \dots \rightarrow G'_{i+j} \rightarrow G'_r) \\
& \text{overHd}(G_1 \rightarrow \dots \rightarrow G_{i+j} \rightarrow G_r \Rightarrow G'_1 \rightarrow \dots \rightarrow G'_i \rightarrow \text{dyn}) \\
& \quad = \text{overHd}(G_1 \rightarrow \dots \rightarrow G_{i+j} \rightarrow G_r \Rightarrow G'_1 \rightarrow \dots \rightarrow G'_i \rightarrow \text{dyn} \rightarrow \dots \rightarrow \text{dyn}) \\
& \text{overHd}([G] \Rightarrow \text{dyn}) = \text{overHd}([G] \Rightarrow [\text{dyn}]) \\
& \text{overHd}(\text{dyn} \Rightarrow [G]) = \text{overHd}([\text{dyn}] \Rightarrow [G]) \\
& \text{overHd}(G_1 \Rightarrow G_2) = \text{createOH}(G_1 \Rightarrow G_2) + \text{callOH}(G_1 \Rightarrow G_2) \\
& \text{createOH}(\text{dyn} \Rightarrow U) = U^p \\
& \text{createOH}(U \Rightarrow \text{dyn}) = U^i \\
& \text{createOH}(G_1 \rightarrow \dots \rightarrow G_r \Rightarrow G'_1 \rightarrow \dots \rightarrow G'_r) = F^c \\
& \text{createOH}([G_1] \Rightarrow [G_2]) = L^c \\
& \text{callOH}(\text{dyn} \Rightarrow U) = 0 \\
& \text{callOH}(U \Rightarrow \text{dyn}) = 0 \\
& \text{callOH}(G_1 \rightarrow \dots \rightarrow G_n \rightarrow G_r \Rightarrow G'_1 \rightarrow \dots \rightarrow G'_n \rightarrow G'_r) \\
& \quad = \sum_1^n \text{overHd}(G'_i \Rightarrow G_i) + \text{overHd}(G_r \Rightarrow G'_r) \\
& \text{callOH}([G_1] \Rightarrow [G_2]) \\
& \quad = \text{overHd}(\text{int} \rightarrow G_1 \Rightarrow \text{int} \rightarrow G_2) + \text{overHd}(\text{int} \rightarrow G_1 \rightarrow \text{unit} \Rightarrow \text{int} \rightarrow G_2 \rightarrow \text{unit})
\end{aligned}$$

 **Figure 5** An overhead gauging algorithm.

similarly delegated to case eight. Case eight deals with all cases not matched by earlier cases. It says that the overhead is an addition of the creation overhead, returned from *createOH*, and the call overhead, returned from *callOH*.

The definition of *createOH* is straightforward: it assigns a corresponding symbolic overhead to each kind of cast. The function *callOH* implements the idea of invocation overheads and access overheads discussed in Section 4. For casts involving base types, the call overhead is 0 because they can not be invoked or no elements may be accessed from them. The call overhead for a function cast is the overhead of casting all parameter types plus that of casting the return type. The call overhead for a list cast is the total overhead of read access and write access.

The following example illustrates the calculation process for gauging the overhead for the cast *dyn* \Rightarrow [*Bool*].

$$\begin{aligned}
& \text{overHd}(\text{dyn} \Rightarrow [\text{Bool}]) \\
& = \text{overHd}([\text{dyn}] \Rightarrow [\text{Bool}]) \\
& = \text{createOH}([\text{dyn}] \Rightarrow [\text{Bool}]) + \text{callOH}([\text{dyn}] \Rightarrow [\text{Bool}]) \\
& = L^c + \text{callOH}([\text{dyn}] \Rightarrow [\text{Bool}]) \\
& = L^c + \text{overHd}(\text{int} \rightarrow \text{dyn} \Rightarrow \text{int} \rightarrow \text{Bool}) + \text{overHd}(\text{int} \rightarrow \text{dyn} \rightarrow \text{unit} \Rightarrow \text{int} \rightarrow \text{Bool} \rightarrow \text{unit}) \\
& = L^c + F^c + \text{overHd}(\text{int} \Rightarrow \text{int}) + \text{overHd}(\text{dyn} \Rightarrow \text{Bool}) + \text{overHd}(\text{int} \rightarrow \text{dyn} \rightarrow \text{unit} \Rightarrow \text{int} \rightarrow \text{Bool} \rightarrow \text{unit}) \\
& = L^c + F^c + 0 + U^p + F^c + \text{overHd}(\text{int} \Rightarrow \text{int}) + \text{overHd}(\text{Bool} \Rightarrow \text{dyn}) + \text{overHd}(\text{unit} \Rightarrow \text{unit}) \\
& = L^c + F^c + 0 + U^p + F^c + U^i \\
& = L^c + 2 \cdot F^c + U^p + U^i
\end{aligned}$$

Due to the limited space, the algorithm in Figure 5 deals with base types, function types, and list types only. Our implementation supports many more types, including dictionary types, tuples, objects, records, and several others, with the same idea.

4.3 Representing Overheads for a Program

Without the loss of generality, we assume that a program consists of a few functions and top-level statements. When the program is translated, casts are inserted into function definitions and top-level statements. To extract the feature vector for a program, we repeat the following for each function. For each cast inserted in the function, we use Figure 5 to calculate the overhead. We then sum the overheads for all casts together. If a cast appears in a loop, then we automatically instrument the loop, obtain the number of times the loop is executed, and multiply the cast overhead by that number. For example, if a function has two casts that are outside of loops and have the overheads $F^c + U^p$ and $L^c + F^c + U^p + U^i$, then the total overhead for that function is $L^c + 2 \cdot F^c + 2 \cdot U^p + U^i$. The feature vector for that function is the coefficients of all overhead symbols, represented as 1, 2, 2, 1 in this case. The machine learning algorithm will turn these coefficients into runtime predictions.

Similarly, for the casts inserted in top-level statements, we calculate the overhead of each cast and sum them together.

Finally, we concatenate representations for all functions and top-level statements, forming a list of coefficients. This list will be the feature representation of the whole program.

4.4 Assessing Feature Effectiveness

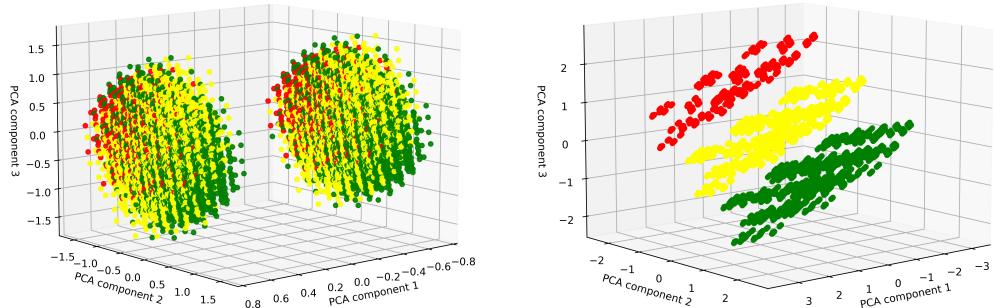


Figure 6 3D PCA analysis for Nbody using bit strings (left) and cast overheads (right). Both figures are generated with elevation of 10.0 and use azimuth angle 50.

Our approach `LearnPerf` is developed using cast overheads as features. For all the benchmarks we used to evaluate the performance, the DR is always less than 4% except for one benchmark whose DR is 5.3% (We will present the results in more detail in Section 5). In general, this means that our predicted time is in average within 4% of difference compared to the real measured time. We view this as a significant improvement over the performance of bit-string based solution, where DR is often higher than 30% and can be as high as 90%.

We have performed a PCA analysis Abdi and Williams [1] to understand the effectiveness of both bit strings and cast overheads. Figure 6 presents the analysis results for Nbody. In the figure, axes represent values of PCA components and colored circles represent configurations. In particular, configurations with similar running times get the same color. The running times of Nbody are roughly in three groups: those less than 7.5s (seconds), between 12.5s and 15s, and more than 20s (see Figure 7 for more details). From Figure 6, we can observe that bit strings fail to separate configurations while cast overheads successfully separate configurations according to their runtimes. Intuitively, clear separations of configurations according to their runtimes mean fewer prediction errors. This shows the usefulness of using cast overheads as features.

Table 3 Python benchmarks used for performance evaluation. The last column gives the number of configurations generated for the corresponding benchmark.

Benchmark	LOC	# of functions	# of pars	# of typed pars	# of configurations
Monte Carlo	90	4	9	9	385
Meteor	238	8	26	14	1024
CPU	2824	32	39	23	2897
Zebrafy	1578	28	72	38	4000
Pascal	70	7	19	15	5184
chaos	271	22	42	29	6000
Richard	455	21	94	67	6000
BenchFirst	1017	27	76	54	6000
Sieve	56	9	22	21	15361
Nbody	195	4	21	18	16384
Scimark	65	5	22	17	25088
Raytrace	254	37	67	38	33904

5 Performance Evaluation

We have implemented `LearnPerf` in Python. The main components are type addition, feature extraction, model training. Some of evaluated benchmarks are adopted from earlier work in gradual typing Campora et al. [8], Vitousek et al. [45], which already have type information. For other benchmarks, we use HiTyper (Peng et al. [30]), a state-of-the-art deep learning approach, to infer types that may be added. One issue with HiTyper is that some inferred types are erroneous, as noted by Yee and Guha Yee and Guha [51]. We remove a type annotation whenever adding it causes static type conflicts. We generate a new, more precisely typed program after merging the type annotations from HiTyper into the original program. From the new program, we generate a desired number of configurations for each benchmark (Table 5).

We implemented feature extractions on top of Reticulated Python (Vitousek et al. [45, 47, 46]). To test the generality of our approach, we have implemented feature extractions for both the guarded semantics Vitousek et al. [45] and transient semantics (Vitousek et al. [47]). Since these two semantics lead to different translated programs, we have different feature extraction codes. However, both implementations are based on the idea of gauging cast overheads, discussed in Section 4. Our feature extraction, which totals about 1,850 lines of code, supports the most commonly used Python types, including lists, functions, dictionary types, tuples, iterables, objects, classes, and many others.

The model training component is implemented on top of the scikit-learn Pedregosa et al. [29] package, a frequently used machine learning Library in Python. We use scikit-learn’s various models, its training-testing data split package, and its metrics package. This component includes less than 200 LOC.

5.1 Benchmarks

To evaluate the performance of `LearnPerf`, we adopted nine benchmarks that were commonly used in gradual typing research in Python (Vitousek et al. [47, 46], Campora et al. [8]. These programs are relatively small, often below 500 LOC. In addition, we adopted three large benchmarks, including Zebrafy (a Python program for creating PDF files) and CPU

Benchmark and BenchFirst (two performance bench-marking programs). For each benchmark, we present the name, lines of code, number of functions, number of parameters, number of parameters that are typed originally or with the help of HiTyper, and total number of configurations we generated for evaluating our performance in Table 3.

The number of configurations generated for each benchmark is mainly determined by two factors: the number of parameters in the benchmark and the time required to run each configuration. For example, each configuration in Zebrafy, CPU Benchmark, and BenchFirst takes more than 100 seconds to finish. As a result, we generate about only 4000 configurations for such benchmarks.

The configurations for each benchmark for evaluating performance are generated follow the insights from Greenman et al. [16] to ensure that they are representative. We can imagine that all configurations from a benchmark be organized into a lattice based on the parameters that are typed. The lattice includes 2^n configurations if n parameters are typed. All configurations in the same row of the lattice assign types to the same number of parameters. For example, the bottom-most row assigns types to zero parameters, and the row above assign types to only one parameters, and the row above that assign types to two parameters, and so on.

In our experiments, we generated configurations such that every row of the lattice is covered. Moreover, we try to maintain same proportion of generated configurations over all configurations in a row across all rows. However, for middle rows, the percentage is smaller because there are too many configurations in them. For example, the middle row has $C_n^{\frac{n}{2}}$ configurations. Once these configurations are generated, we randomly split them so that 40 are used for training and the remaining are used for testing. Note, we repeated 5 times for the training/testing process.

The running times in this paper are measured on a machine equipped with Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz, 8 Core(s), and 16GB RAM. Each measured time is an average of 10 runs.

Figure 7 gives an idea of how execution times look as a certain number of parameters are typed. The figure shows that while the running times of some benchmarks are clustered, others are scattered. We believe that these benchmarks serve the evaluation purpose well.

Our evaluation focus on Scenarios 1 and 3 only. The result for Scenario 2 is similar to that for Scenario 3, and we omit it in the paper.

5.2 Supporting Scenario 1

To simulate the real development scenario, we randomly selected 40 instances from all generated configurations as training instances, and we use linear regression to train a time prediction model. Compared to standard machine learning applications, our approach uses significantly fewer data instances for training.

To ensure that the model correctly learns patterns from the data and doesn't pick up too much noise, we used k-fold cross-validation technique. As is standard in machine learning practice, our results are averaged over all k trials to get the overall performance of the model. We set k to 5 in our evaluation. Experiments were run on the same machine we used for generating benchmark's configurations.

Table 4 describes the performance of `LearnPerf` on all evaluated benchmarks. Columns three through five of the table show that even when the model is trained with only 40 instances, our prediction result is very accurate, with DR (defined in Section 3.2) less than 3% for nine benchmarks, between 3% and 4% for two benchmarks, and is 5.26% for one benchmark. Intuitively, this means that our predicted times are very close to measured times.

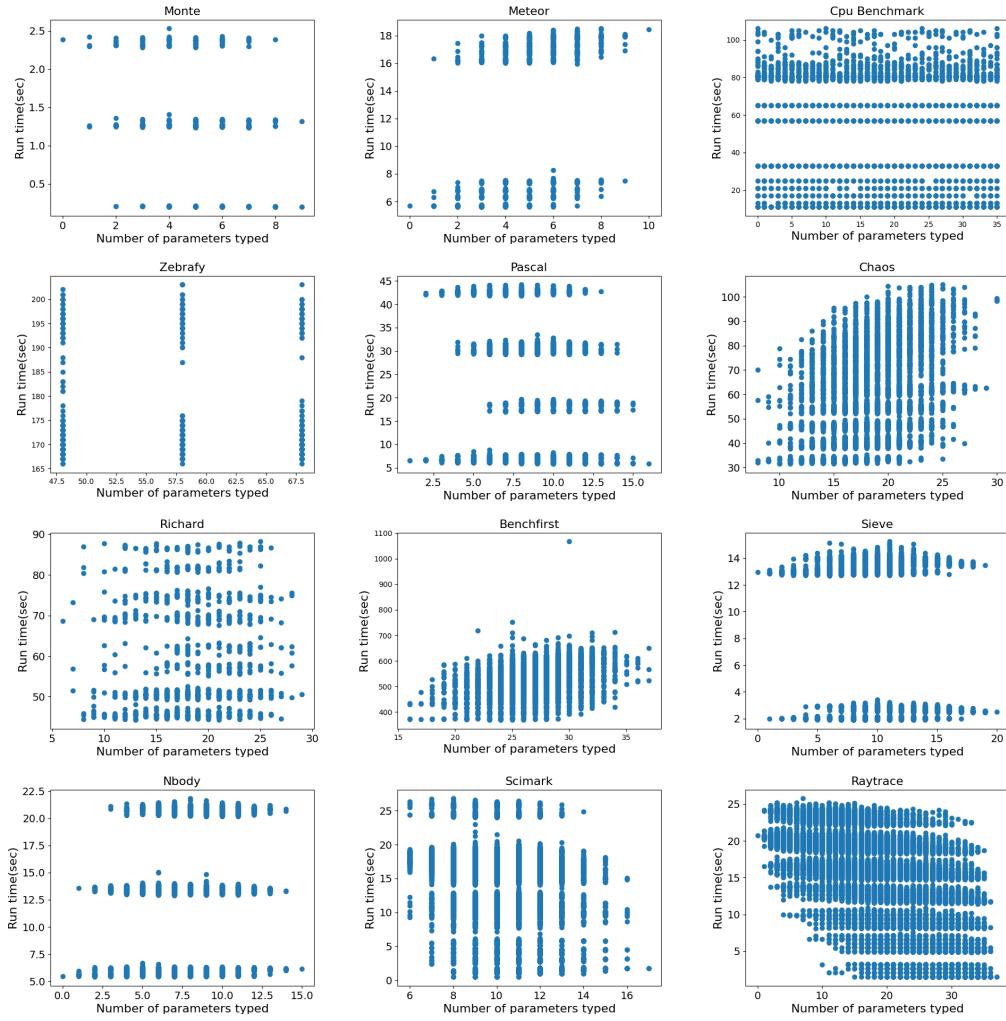


Figure 7 Benchmark's configurations description: Run Time vs Number of parameters typed for each benchmark.

Columns six and seven of Table 4 present the ratios of configurations whose *DR* are less and greater than 10%, respectively. The result shows that there are fewer configurations that have large difference ratios.

Figure 8 presents a closer investigation of the evaluation result. Specifically, we divide each benchmark into five groups in terms of their measured running time of different configurations of the benchmark. Next, we predict the performance (running time) and measure the *DR* of all configurations within each group. For every group, green represents a *DR* of less than 5%, cyan represents 5 to 10%, blue represents 10 to 15%, violet represents 15 to 20%, and red represents more than 20% of *DR*. The figure reveals that, in general, the configurations that have smaller running times tend to experience higher *DRs*. There are two potential reasons behind this. First, a small variance in predicted time for such configurations can lead to a higher *DR*. Second, even averaged over ten runs, each measured time includes a small randomness due to computer execution dynamics, and the randomness in such configurations has a more conspicuous impact.

Table 4 The performance of `LearnPerf` on evaluated benchmarks. The model for each benchmark is trained with forty randomly selected configurations. The second column gives the number of testing instances (configurations). Columns three through five gives the average performance of all testing instances. Columns six and seven give the ratios of instances whose *DR* are less than and greater than 10%, respectively.

Benchmark	# testing	MAE	MSE	<i>DR</i>	<10%	>10%
Monte Carlo	344	0.05 ± 0.01	0.07 ± 0.01	3.597%	93.77%	6.23%
Meteor	984	0.32 ± 0.01	0.39 ± 0.02	2.70 %	97.35%	2.65%
CPU	2857	1.56 ± 0.01	2.07 ± 0.08	1.91%	99.80%	0.2%
Zebrafy	3960	2.70 ± 0.00	3.63 ± 0.00	1.57%	92.88%	7.12%
Pascal	5144	0.37 ± 0.03	0.460 ± 0.04	2.10%	95.55%	4.45
Chaos	5960	2.37 ± 0.07	2.92 ± 0.09	3.56%	97.09%	2.91%
Richard	5960	0.55 ± 0.00	0.71 ± 0.00	0.91%	100%	0.0%
BenchFirst	5960	17.12 ± 0.03	25.06 ± 0.7	5.26%	86.04%	13.96%
Sieve	15321	0.17 ± 0.01	0.23 ± 0.01	2.18%	89.06%	10.94%
Nbody	16344	0.21 ± 0.01	0.25 ± 0.01	1.84%	99.86%	0.14%
Scimark	25048	0.14 ± 0.04	0.193 ± 0.05	0.97%	98.92%	1.08%
Raytrace	33864	0.26 ± 0.06	0.330 ± 0.09	1.73%	94.46%	5.54%

5.3 Supporting Scenario 3

Scenario 3 aims to find the neighbor with best performance for any given configuration. This is particularly helpful when the current configuration has poor performance and the user wants to find a neighbor with good performance. We can imagine that there are two lattices with the current configuration. One grows up, adding more type annotations to current configuration, and one grows down, removing type annotations from the current configuration. We then use the idea of these two method to choose neighbors. However, here we consider configurations that add/remove up to seven parameters. To evaluate how well `LearnPerf` can support this scenario, we randomly choose a certain number of configurations, and find the most performant neighbor of it using our model.

We present the detailed result for this scenario in Table 5, which includes the number of configurations considered as the current configuration (the second column) and three metrics to measure the performance of `LearnPerf`. To simplify our discussion below, we refer to a configuration and all its neighbors as a *region*. Each region includes at least 100 neighbors or includes all neighbors that add types to up to seven parameters. The first metric (column three in the table) is the accuracy. For any given configuration, if the most performant neighbor identified by `LearnPerf` is among the three neighbors with least execution times, then we classify this as a correct identification. We consider top three neighbors because it is common for many neighbors to have very small difference in execution times. The accuracy is calculated by dividing the number of correct identifications over all regions considered for that benchmark. For example, for Scimark, we considered 500 regions, and for 408 regions `LearnPerf` made correct identifications. As a result, the accuracy is 81.6%.

The second metric (column four in the table) is the average differences between the execution times of the real and the identified most performant neighbors. For example, if the real most performant neighbor for a region has an execution time of 4.73s and the identified neighbor has an execution time of 4.75s, then the time difference is 0.02s. This column records the average of differences of all regions for that benchmark. The third metric (column five in the table) calculates the time difference in percentage.

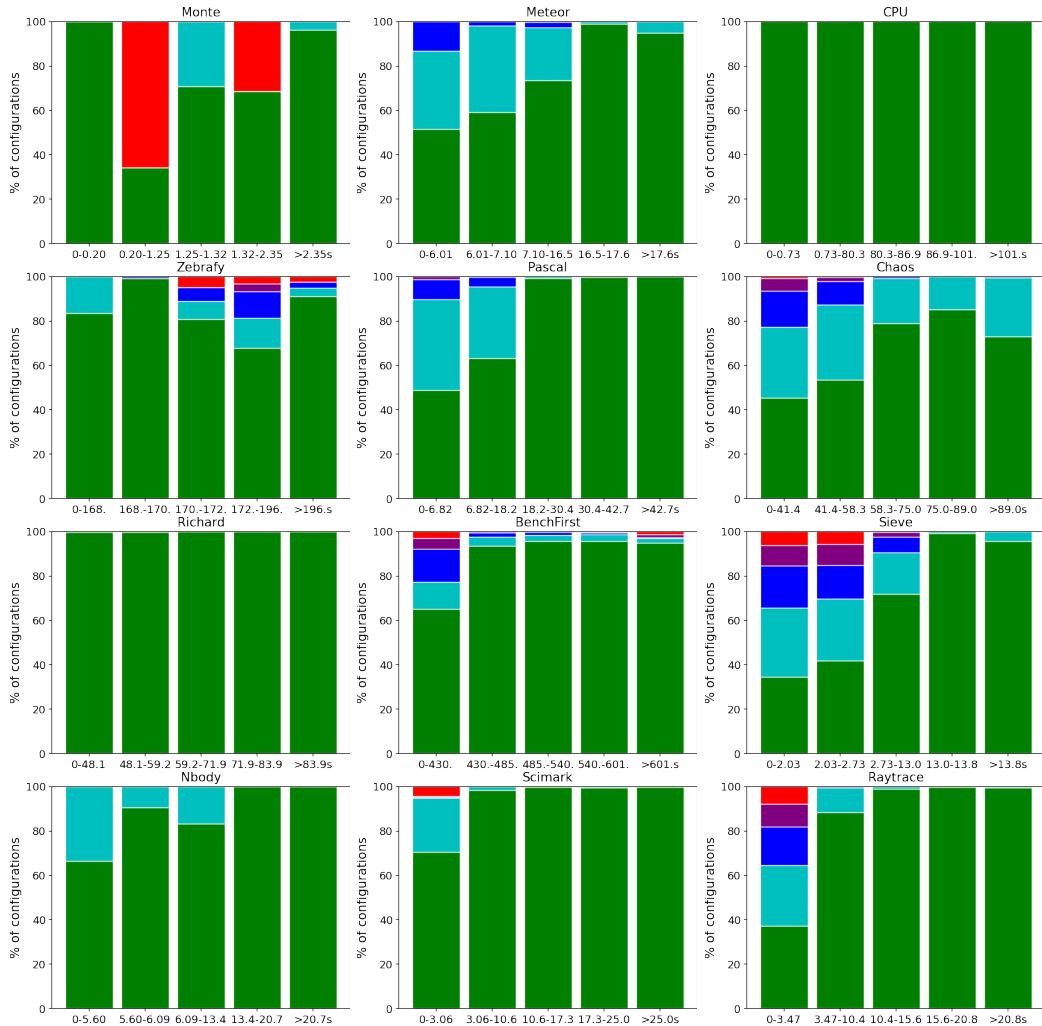


Figure 8 LearnPerf detailed benchmark's performance based on different measured run time groups.

Again, the table shows that our approach is very accurate in identifying the most performant neighbors, with the difference ratio always below 1% except for Pascal that has a 2.4% DR.

5.4 Training and Prediction Times

Table 6 presents times needed for generating and measuring 40 configurations for training the model, the time for training the model once these 40 configurations are ready, and the average feature extraction time for each program. We do not present the prediction time because that is less than 1ms for each configuration. From the table, we can see that the most time in our approach is spent on measuring the running times for training the model.

For some benchmarks, measuring the times is relatively fast, such as for Monte Carlo, Meteor, Sieve, Nbody, Scimark, and Raytrace. However, it takes significantly longer to measure the times for some benchmarks, including CPU, Zebrafy, Chaos, Richard, and BenchFirst. The reason is that each configuration from these benchmarks takes a long time to complete. Usually, this large amount of measuring time will lead to a long response time. Also, it looks like this long waiting time is hard to avoid.

21:20 Learning Gradual Typing Performance

Table 5 LearnPerf's performance on finding the most performant neighbor to migrate for each benchmark.

Benchmark	# of regions	Accuracy	Average difference(s)	difference ratio
Monte Carlo	42	100%	0.0	0%
Meteor	500	77.0%	0.032	0.338%
CPU	38	94.74%	0.004	0.998%
Zebrafy	89	98.88%	0.007	0.087%
Pascal	500	44.80%	0.171	2.388%
chaos	297	83.16%	0.059	0.88%
Richard	98	100%	0	0%
BenchFirst	113	93.81%	0.004	0.058%
Sieve	500	63.6%	0.020	0.685%
Nbody	500	34.60%	0.116	1.246%
Scimark	500	81.6%	0.021	0.344%
Raytrace	385	96.88%	0.007	0.034%

Table 6 Training and Prediction time of Each benchmark.

Benchmark	Measuring 40 configurations (s)	Training(s)	Feature extraction (ms)
Monte Carlo	53.27	1.00	10.98
Meteor	490.06	0.99	23.38
CPU	2997.87	3.3	1001.96
Zebrafy	7394.38	4.75	1012.30
Pascal	580.05	1.01	40.89
Chaos	2654.87	1.03	29.15
Richard	2462.77	1.99	1013.33
BenchFirst	21816.94	3.89	1112.32
Sieve	373.33	1.02	19.67
Nbody	488.94	0.99	25.86
Scimark	555.68	0.99	27.73
Raytrace	623.55	2.98	26.17

Fortunately, with the help of type migration tools, we can significantly shorten the response time. The idea is that we start to measure the runtimes way before the user needs the migration support. We tested this idea by automating the process of generating type information for parameters with HiTyper, merging the generated type information into the original program, randomly generating configurations for training, running all generated configurations to measure their runtime duration, extracting features for these configurations, and training a time prediction model based on the collected times and extracted features. We tested this idea on three large benchmarks, including CPU, Zebrafy, and BenchFirst.

Once the model has been trained, predicting the running time is very fast. Since feature generation is also very efficient, we can quickly provide migration support with the model. For example, for any given configuration, LearnPerf is able to find the most performant neighbor within a few seconds.

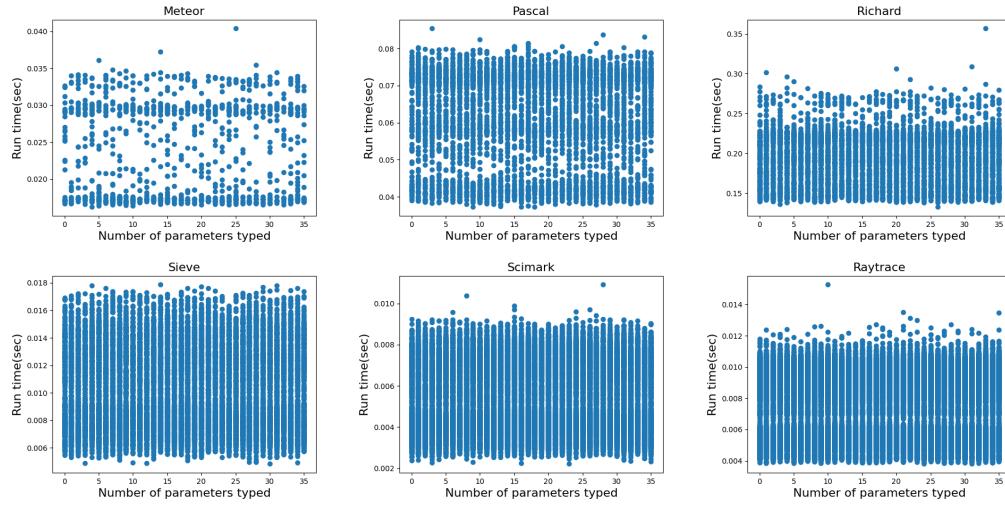


Figure 9 The relation between run times and the number of parameters that have type annotations for six benchmarks for transient semantics.

5.5 Different Machine Learning Methods

We used linear regression to train our model. During the development of `LearnPerf`, we also explored other machine learning algorithms, including random forest regression, decision tree regression, and AdaBoost regression. We decided to use linear regression for the following reasons. First, linear regression usually does not need too many training instances. In our case, 40 training instances yield good performance. Second, training and prediction with linear regression is very fast than other models. Third, linear regression yields good performance across all benchmarks. For example, while random forest achieves 1.38% and 1.19% *DRs* for Monte and Sieve, respectively, the *DRs* for Raytrace and Scimark are above 13%.

We also tried some other famous Machine learning models, such as support vector machine regression and MLP regression, but they either need more training instance or take more times for training and prediction. Also, they do not outperform linear regression for our problem.

5.6 Evaluation of Transient Semantics

In addition to evaluating the performance of `LearnPerf` on the guarded semantics, we have also evaluated it on the transient semantics (Vitousek et al. [47]) using the same benchmarks. In our feature extraction code for transient semantics, we set all the return values of *callOH*(\cdot) in Figure 5 to 0 because transient casts do not introduce proxies.

Figure 9 presents the runtime distributions for six benchmarks under transient semantics. We omit the other six because their distributions are similar. Comparing this figure to Figure 7 we observe that the runtimes in transient semantics are several magnitudes smaller than in guarded semantics. Also, the runtimes have smaller variations across different configurations.

Table 7 presents the performance of `LearnPerf` for the transient semantics. We can observe that the *DR* is much higher than that for guarded semantics. Meanwhile, we observe that the *MAE* and *MSE* are close to 0. This indicates that a possible reason that *DR* is relatively high because the runtime of each configuration is very smaller, usually below 0.1 seconds. A small randomness in measured time can lead to a high *DR* in this case.

 **Table 7** Overall performance of `LearnPerf` for the transient semantics.

Benchmark	# training	# testing	MAE	MSE	DR
Monte Carlo	40	344	0.002 ± 0.0	0.004 ± 0.0	31.43%
Meteor	40	984	0.005 ± 0.01	0.006 ± 0.0	25.09 %
CPU	40	2857	0.01 ± 0.0	0.02 ± 0.08	18.91%
Zebrafy	40	3960	0.001 ± 0.0	0.003 ± 0.0	27.77%
Pascal	40	5144	0.01 ± 0.0	0.013 ± 0.0	18.549%
Chaos	40	5960	0.02 ± 0.0	0.06 ± 0.0	34.38%
Richard	40	5960	0.027 ± 0.0	0.033 ± 0.0	24.93%
BenchFirst	40	5960	0.033 ± 0.0	0.087 ± 0.0	25.88%
Sieve	40	15321	0.002 ± 0.0	0.001 ± 0.0	29.07%
Nbody	40	16344	0.003 ± 0.0	0.005 ± 0.01	21.78%
Scimark	40	25048	0.001 ± 0.0	0.001 ± 0.0	27.129%
Raytrace	40	33864	0.001 ± 0.0	0.002 ± 0.0	25.497%

Overall, our approach works pretty well for transient semantics also. The main insight is that the algorithm in Figure 5 derives coefficients of cast overheads rather than the real runtime overhead of casts. The overhead of a transient cast (checking type tags) can also be estimated using coefficients.

5.7 Threats to Validity

It may be possible that the results observed in our evaluation do not transfer to other Python programs. We have done the following to reduce this possibility. (1) We chose the benchmarks that are commonly used in the literature for evaluating gradual typing performance as well as three large Python programs (details in Section 5.1). (2) The evaluated programs cover most commonly used language features in Python, including control structures such as conditionals and loops, functions, classes with inheritance, tuples, dictionaries, nested lists, etc. (3) The amount of typed parameters can have a big impact on the results. As shown in Table 5, the percentages that parameters have types are quite diverse, ranging from about 50% (Meteor and Zebrafy) to about 100% (Monte Carlo, Sieve, and Nbody). (4) The kinds of casts in translated programs could also affect the performance of `LearnPerf`. After checking the translated programs, we observed the presence of simple casts (about 63% of all casts) between basic types as well as higher-order casts (about 37% of all casts) between function types, list types, and object types. (5) Each time is an average of 10 runs and each machine learning experiment is averaged over 5 trials.

6 Related Work

Understanding performance changes during migration. While a lot of work has been done to automatically migrate dynamic programs toward more static, little work has been done to aid the performance aspect during program migration except for a few efforts.

Our work is closely related to the work by Campora et al. [8]. They developed HERDER to help navigate the performance landscape during migration. However, there are many differences between `LearnPerf` and HERDER. First, `LearnPerf` is able to precisely predict a time for any configuration while HERDER is able to find only a symbolic overhead for each configuration. There is no direct mapping from these symbolic values to real runtimes

and so the relation between two symbolic values often does not carry over to the real runtimes. For example, two configurations from a single benchmark have symbolic values $2 * \ell_3 * \ell_4$ and $67 * \ell_3 * \ell_4$, respectively, while their corresponding runtimes are 24.79 and 37.38 seconds, respectively. As a result, several migration supports are possible with `LearnPerf` but not `HERDER`, such as classifying neighbors of a certain configuration based on their speedup/slowdown ratios.

Another difference is that, since our approach is based on machine learning, we only need to extract approximate values for features. `HERDER`, however, is based on static analysis and needs to be very accurate. For example, in `LearnPerf`, the overhead for a function cast is a simple addition of creation overhead and invocation overhead. In `HERDER`, a function cast needs to be transformed to an intermediate language to simulate the creation of proxies. As a result, it is easier to support more language features in `LearnPerf` than in `HERDER`. For example, we support object and class types, but they were missing in `HERDER`.

Greenman et al. [15] also investigated the performance problem during program migration but from a very different perspective. Through a large-scale empirical study, the authors studied how outputs from profilers may be exploited for proving migration supports. They considered seventeen strategies for how to avoid configurations with unacceptable performance and navigate to configurations to acceptable performance. Through the study, they generated three useful lessons for developers and one lesson for language designers for how to deal with the performance problem. Their focus is very different from our work in that we aim to predict the runtime for each configuration, and provide other migration supports, such as finding the best performing configuration among the neighbors, on top of that.

Assessing and Optimizing Gradual Typing Performance. Takikawa et al. [41] evaluated the performance of Typed Racket, focusing on the areas mixing untyped and typed code. The evaluation revealed significant runtime overhead in sound gradual typing. In evaluating the performance of a gradual type system, Greenman et al. [16] conducted a thorough analysis by fully annotating a series of benchmarks in Typed Racket. Absolute performance calculations were derived by generating a significant subset of configurations from the complete lattice of possible configurations. Performance ratios for each configuration were then compared against base configurations to identify K-step and D-deliverable configurations.

Since the report of the performance problem in gradual typing, a lot of work has been done to solve this problem, ranging from designing new type systems or new languages, inferring more types to reduce casts, to developing more efficient cast languages.

Rastogi et al. [34] introduced a type inference algorithm for existing gradually typed code, especially focusing on the inflow and outflow of types. Their approach supports open-world soundness to enable sound interactions with unseen code. Instead, Nguyen et al. [27] used static analysis to remove casts that always succeed without considering open-world soundness.

The idea of developing new languages to avoid expensive interactions has been explored by Muehlboeck et al. Muehlboeck and Tate [26]. Several approaches have been developed to exploit compilers or JITs to improve gradual typing performance Rastogi et al. [35], Richards et al. [36], Bauman et al. [5]. Another important line of improving gradual typing performance is through the design of new or change cast constructs Feltey et al. [12], Kuhlenschmidt et al. [20]. The work by Allende et al. [4] designed confined gradual typing, allowing users to control the flowing of type information through type annotations for reducing expensive boundary crossings.

Our approach is complementary to these approaches in that they do not try to compare the performance of different configurations and identify performant configurations. Also, while these approaches optimize the performance of gradual programs, they often do not

fully reduce the overheads due to runtime type checks. This paper shows that our approach works well for both the guarded and transient semantics. It looks promising in applying our idea to the translated programs from these approaches to predict the performance of these optimized programs.

Machine Learning for Programming. Many machine learning based approaches have been developed for solving programming language and software engineering problems Wan et al. [49], Allamanis et al. [2]. A main trend is using deep models, such as large language models, to automatically extract code features. Interestingly, our exploration shows that deep learning approach does not produce a good model for performance prediction for our problem. Vo and Nguyen [48] observed a similar phenomenon for vulnerability detection.

7 Conclusion

With gradual typing, developers enjoy the benefits of both static and dynamic typing. A major obstacle of adopting gradual typing is that the runtime overhead when going from less typed regions to more typed regions is often high and unpredictable. To address this issue, we developed a machine learning-based solution named `LearnPerf` that approximates runtime overheads due to inserted casts. We have evaluated our approach on 12 Python benchmarks, with each of the three large benchmarks having more than 1000 LOC. The evaluation results demonstrated that `LearnPerf` is able to precisely predict the execution time of each configuration. On top of that, we can develop further migration supports, such as finding the most performant neighbor of a configuration when it has poor performance. Our approach works well for both guarded and transient semantics. In the future, we would like to extend our approach to support a more macro level gradually-typed language, such as Typed Racket. It is also interesting to investigate if our approach can be employed to predict the performance of optimized gradual programs.

References

- 1 Hervé Abdi and Lynne J Williams. Principal component analysis. *Wiley interdisciplinary reviews: computational statistics*, 2(4):433–459, 2010.
- 2 Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51(4), July 2018. doi:10.1145/3212695.
- 3 Miltiadis Allamanis, Earl T. Barr, Soline Ducouso, and Zheng Gao. Typilus: neural type hints. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, June 2020. doi:10.1145/3385412.3385997.
- 4 Esteban Allende, Johan Fabry, Ronald Garcia, and Éric Tanter. Confined gradual typing. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA ’14, pages 251–270, New York, NY, USA, 2014. ACM. doi:10.1145/2660193.2660222.
- 5 Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy Siek, and Sam Tobin-Hochstadt. Sound gradual typing: Only mostly dead. *Proc. ACM Program. Lang.*, 1(OOPSLA):54:1–54:24, October 2017. doi:10.1145/3133878.
- 6 John Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw. Migrating gradual types. In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL ’18, New York, NY, USA, 2018. ACM.
- 7 John Peter Campora and Sheng Chen. Taming type annotations in gradual typing. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. doi:10.1145/3428259.

- 8 John Peter Campora, Sheng Chen, and Eric Walkingshaw. Casts and costs: Harmonizing safety and performance in gradual typing. *Proc. ACM Program. Lang.*, 2(ICFP):98:1–98:30, July 2018. doi:[10.1145/3236793](https://doi.org/10.1145/3236793).
- 9 Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. Gradual typing: A new perspective. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:[10.1145/3290329](https://doi.org/10.1145/3290329).
- 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. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 410–429, New York, NY, USA, 2016. ACM. doi:[10.1145/2983990.2984017](https://doi.org/10.1145/2983990.2984017).
- 11 Fernando Cristiani and Peter Thiemann. Generation of typescript declaration files from javascript code. In *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR 2021, pages 97–112, New York, NY, USA, 2021. Association for Computing Machinery. doi:[10.1145/3475738.3480941](https://doi.org/10.1145/3475738.3480941).
- 12 Daniel Feltey, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour. Collapsible contracts: Fixing a pathology of gradual typing. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi:[10.1145/3276503](https://doi.org/10.1145/3276503).
- 13 Ronald Garcia and Matteo Cimini. Principal type schemes for gradual programs. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’15, pages 303–315, New York, NY, USA, 2015. ACM. doi:[10.1145/2676726.2676992](https://doi.org/10.1145/2676726.2676992).
- 14 Ben Greenman and Matthias Felleisen. A spectrum of type soundness and performance. *Proc. ACM Program. Lang.*, 2(ICFP):71:1–71:32, July 2018. doi:[10.1145/3236766](https://doi.org/10.1145/3236766).
- 15 Ben Greenman, Matthias Felleisen, and Christos Dimoulas. How profilers can help navigate type migration. *Proc. ACM Program. Lang.*, 7(OOPSLA2), October 2023. doi:[10.1145/3622817](https://doi.org/10.1145/3622817).
- 16 Ben Greenman, Asumu Takikawa, Max S. New, Daniel Feltey, Robert Bruce Findler, Jan Vitek, and Matthias Felleisen. How to evaluate the performance of gradual type systems. *Journal of Functional Programming*, 29:e4, 2019. doi:[10.1017/S0956796818000217](https://doi.org/10.1017/S0956796818000217).
- 17 Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint*, 2022. arXiv:2203.03850.
- 18 Sergiy Kolesnikov, Norbert Siegmund, Christian K’astner, Alexander Grebhahn, and Sven Apel. Tradeoffs in modeling performance of highly configurable software systems. *Software & Systems Modeling*, 18:2265–2283, June 2019. doi:[10.1007/s10270-018-0662-9](https://doi.org/10.1007/s10270-018-0662-9).
- 19 Erik Krogh Kristensen and Anders Møller. Type test scripts for typescript testing. *Proc. ACM Program. Lang.*, 1(OOPSLA):90:1–90:25, October 2017. doi:[10.1145/3133914](https://doi.org/10.1145/3133914).
- 20 Andre Kuhlenschmidt, Deyaa 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, pages 517–532, New York, NY, USA, 2019. Association for Computing Machinery. doi:[10.1145/3314221.3314627](https://doi.org/10.1145/3314221.3314627).
- 21 Andréa Matsunaga and José A.B. Fortes. On the use of machine learning to predict the time and resources consumed by applications. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 495–504, 2010. doi:[10.1109/CCGRID.2010.98](https://doi.org/10.1109/CCGRID.2010.98).
- 22 Zeina Migeed and Jens Palsberg. What is decidable about gradual types? *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi:[10.1145/3371097](https://doi.org/10.1145/3371097).
- 23 Amir M Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. Type4py: practical deep similarity learning-based type inference for python. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2241–2252, 2022.
- 24 Yusuke Miyazaki, Taro Sekiyama, and Atsushi Igarashi. Dynamic type inference for gradual hindley–milner typing. *Proc. ACM Program. Lang.*, 3(POPL):18:1–18:29, January 2019. doi:[10.1145/3290331](https://doi.org/10.1145/3290331).

- 25 Cameron Moy, Phúc C. Nguyễn, Sam Tobin-Hochstadt, and David Van Horn. Corpse reviver: Sound and efficient gradual typing via contract verification. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. doi:10.1145/3434334.
- 26 Fabian Muehlboeck and Ross Tate. Sound gradual typing is nominally alive and well. In *OOPSLA*, New York, NY, USA, 2017. ACM. doi:10.1145/3133880.
- 27 Phúc C. Nguyen, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. Soft contract verification for higher-order stateful programs. *Proc. ACM Program. Lang.*, 2(POPL):51:1–51:30, December 2017. doi:10.1145/3158139.
- 28 Francisco Ortín, Miguel García, and Seán McSweeney. Rule-based program specialization to optimize gradually typed code. *Knowledge-Based Systems*, 179:145–173, 2019. doi:10.1016/j.knosys.2019.05.013.
- 29 F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- 30 Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. Static inference meets deep learning: A hybrid type inference approach for python. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE ’22, pages 2019–2030, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3510003.3510038.
- 31 Luna Phipps-Costin, Carolyn Jane Anderson, Michael Greenberg, and Arjun Guha. Solver-based gradual type migration. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021. doi:10.1145/3485488.
- 32 Marius-Constantin Popescu, Valentina E Balas, Liliana Perescu-Popescu, and Nikos Mastorakis. Multilayer perceptron and neural networks. *WSEAS Transactions on Circuits and Systems*, 8(7):579–588, 2009.
- 33 Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. Typewriter: Neural type prediction with search-based validation, 2020. arXiv:1912.03768.
- 34 Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. The ins and outs of gradual type inference. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’12, pages 481–494, New York, NY, USA, 2012. ACM. doi:10.1145/2103656.2103714.
- 35 Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin M. Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for typescript. In *POPL*, 2015.
- 36 Gregor Richards, Ellen Arteca, and Alexi Turcotte. The vm already knew that: Leveraging compile-time knowledge to optimize gradual typing. *Proc. ACM Program. Lang.*, 1(OOPSLA):55:1–55:27, October 2017. doi:10.1145/3133879.
- 37 Jeremy Siek, Ronald Garcia, and Walid Taha. Exploring the design space of higher-order casts. In Giuseppe Castagna, editor, *Programming Languages and Systems*, pages 17–31, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- 38 Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *In Scheme and Functional Programming Workshop*, pages 81–92, 2006.
- 39 Jeremy G. Siek and Manish Vachharajani. Gradual typing with unification-based inference. In *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS ’08, pages 7:1–7:12, New York, NY, USA, 2008. ACM. doi:10.1145/1408681.1408688.
- 40 Dagmar Stumpfe and Jürgen Bajorath. Exploring activity cliffs in medicinal chemistry. *Journal of Medicinal Chemistry*, 55(7):2932–2942, 2012. PMID: 22236250. doi:10.1021/jm201706b.
- 41 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 ’16, pages 456–468, New York, NY, USA, 2016. ACM. doi:10.1145/2837614.2837630.
- 42 Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: From scripts to programs. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA ’06, pages 964–974, New York, NY, USA, 2006. ACM. doi:10.1145/1176617.1176755.

- 43 Sam Tobin-Hochstadt, Matthias Felleisen, Robert Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa. Migratory Typing: Ten Years Later. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, volume 71 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:17, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.SNAPL.2017.17.
- 44 Preston Tunnell Wilson, Ben Greenman, Justin Pombrio, and Shriram Krishnamurthi. The behavior of gradual types: A user study. In *DLS*, number ICFP in DLS 2018, page 1–12, 2018. doi:10.1145/3393673.3276947.
- 45 Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. Design and evaluation of gradual typing for python. In *Proceedings of the 10th ACM Symposium on Dynamic Languages*, DLS ’14, pages 45–56, New York, NY, USA, 2014. ACM. doi:10.1145/2661088.2661101.
- 46 Michael M. Vitousek, Jeremy G. Siek, and Avik Chaudhuri. Optimizing and evaluating transient gradual typing. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2019, pages 28–41, New York, NY, USA, 2019. ACM. doi:10.1145/3359619.3359742.
- 47 Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. Big types in little runtime: Open-world soundness and collaborative blame for gradual type systems. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 762–774, New York, NY, USA, 2017. ACM. doi:10.1145/3009837.3009849.
- 48 Hieu Dinh Vo and Son Nguyen. Can an old fashioned feature extraction and a light-weight model improve vulnerability type identification performance? *arXiv preprint*, 2023. arXiv:2306.14726.
- 49 Yao Wan, Yang He, Zhangqian Bi, Jianguo Zhang, Hongyu Zhang, Yulei Sui, Guandong Xu, Hai Jin, and Philip S Yu. Deep learning for code intelligence: Survey, benchmark and toolkit. *arXiv preprint*, 2023. arXiv:2401.00288.
- 50 Jun Xia, Lecheng Zhang, Xiao Zhu, and Stan Z. Li. Why deep models often cannot beat non-deep counterparts on molecular property prediction? In *ICML 3rd Workshop on Interpretable Machine Learning in Healthcare (IMLH)*, 2023. URL: <https://openreview.net/forum?id=hJG8xgj2Y5>.
- 51 Ming-Ho Yee and Arjun Guha. Do machine learning models produce typescript types that type check? *arXiv preprint*, 2023. arXiv:2302.12163.

Constrictor: Immutability as a Design Concept

Elad Kinsbruner¹  

Technion, Haifa, Israel

Shachar Itzhaky  

Technion, Haifa, Israel

Hila Peleg  

Technion, Haifa, Israel

Abstract

Many object-oriented applications in algorithm design rely on objects never changing during their lifetime. This is often tackled by marking object references as read-only, e.g., using the `const` keyword in C++. In other languages like Python or Java where such a concept does not exist, programmers rely on best practices that are entirely unenforced. While reliance on best practices is obviously too permissive, const-checking is too restrictive: it is possible for a method to mutate the *internal state* while still satisfying the property we expect from an “immutable” object in this setting. We would therefore like to enforce the immutability of an object’s *abstract state*.

We check an object’s immutability through a *view* of its abstract state: for instances of an immutable class, the view does not change when running any of the class’s methods, even if some of the internal state does change. If all methods of a class are verified as non-mutating, we can deem the entire class view-immutable. We present an SMT-based algorithm to check view-immutability, and implement it in our linter/verifier, CONSTRCTOR.

We evaluate CONSTRCTOR on 51 examples of immutability-related design violations. Our evaluation shows that CONSTRCTOR is effective at catching a variety of prototypical design violations, and does so in seconds. We also explore CONSTRCTOR with two real-world case studies.

2012 ACM Subject Classification Software and its engineering → Software design engineering; Software and its engineering → Software defect analysis

Keywords and phrases Immutability, Design Enforcement, SMT, Liskov Substitution Principle, Object-oriented Programming

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.22

Supplementary Material Software (ECOOP 2024 Artifact Evaluation approved artifact):

<https://doi.org/10.4230/DARTS.10.2.9> [36]

Software (ECOOP 2024 Artifact Evaluation approved artifact): <https://doi.org/10.5281/zenodo.11003108>

Funding This research was supported by the Israeli Science Foundation (ISF) grants no. 2117/23 and 651/23.

1 Introduction

Object-oriented code routinely manipulates objects and passes around references to them, some of which are stored in other objects. Parts of the code often rely on some object not being changed during its lifetime. This may be in order to uphold some properties as thread safety [30], security [50] and the stability of invariants [31], allow the use of features like interning [11], or improve the readability of the code [24]. Other considerations include information leakage [50] and concurrency [30]. For these reasons, client code may be written under the assumption that objects on which it relies do not change.

¹ Corresponding author.

 © Elad Kinsbruner, Shachar Itzhaky, and Hila Peleg;
licensed under Creative Commons License CC-BY 4.0
38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 22; pp. 22:1–22:29



22.2 Constrictor: Immutability as a Design Concept

In each of these use cases, the term *immutability* denotes some set of specific assumptions about the object that the use case requires: when used from multiple threads, an object's fields must be available to read without data races; to be safe to pass into an API, the client programmer wants to know foreign code is not going to break the API's relied-upon invariants; when used as a key in a hash table, the library assumes that the hash value of the object is going to remain constant. Despite different needs relying on different assumptions, programming practices rely on one of two solutions: (i) documentation-based agreements at the project or language level [44] that delegate all responsibility to human users, or (ii) annotations that can be checked by the compiler or some external tool.

The first option is extremely expressive – as expressive as humans are – but has the obvious downside of the risk of human error. In the scope of checked annotations, some language features provide some steps in this direction: C++'s `const` keyword and Java's `final` that designate fields and variables as read-only. However, neither of these is a good match for the cases described above: `final` only blocks assignment to a field or variable, but the referenced object can still be mutated via function calls, and `final` does not provide guarantees about object fields unless those happen to be `final` as well.

C++'s `const` is seemingly a better fit, but is still not expressive enough. First, a similar problem to `final` still exists, where a pointer/reference being `const` and its content being immutable are still managed separately (e.g., `const A* const`), and that decision is still left to the programmer. In addition to that, `const` can be used on a specific method, indicating that the method cannot mutate any fields. This does not allow declaring an entire interface as immutable, only certain method; and other methods, particularly ones introduced via inheritance, can mutate any field, including those accessed by `const` methods. The semantics of `const` cannot be used to enforce the property that an interface and all its implementing hierarchy be immutable. In addition to this, for some use-cases it is too restrictive not to be able to assign to *any* field, so C++ also allows marking fields as `mutable` (can be changed even from `const` methods). It is then, again, the user's responsibility to use this annotation responsibly, and no formal guarantees are provided by the compiler.

Immutability in class hierarchies. When provided with an interface or class that is supposed to be immutable, a programmer would like to take advantage of this immutability for purposes of design simplicity or for various optimizations. However, implementing classes and subclasses can introduce unwanted mutations. Languages like Java and C# handle this by marking key library classes (e.g., strings) as `final` or `sealed`. This still does not protect the user from contract mismatches within the library implementation; and, moreover, it precludes legitimate extensions of classes in ways that do not violate the immutability guarantees. This is one of the instances for which the Liskov Substitution Principle (LSP) [39] applies; inheritance as a language mechanism cannot enforce the preservation of properties, and lack of mutations is one such property. The LSP is a *principle*, rather than a *mechanism*, because it is not always possible to distinguish implementations that preserve the properties and ones that do not; and because the properties themselves are often implicit.

Kotlin collections are an interesting example – Figure 1 shows a truncated version of two interfaces, `List` and `MutableList`, from the Kotlin standard library. As summed up by a Google developer [37]:

MutableList, as the name implies, is a list that has operations to mutate, or change, its contents: add, remove, and replace items. It's easy to come to the conclusion that the List type must therefore be immutable. That's not the case. Lists are "read-only", but they may or may not be mutable. [...] The MutableList interface extends the List interface, so it's very easy to create a list that you can change, but pass it around to other code so that code can only read it, even as you're still making changes.

```

interface List<E> {
    operator fun get(index: Int): E
    fun indexOf(element: E): Int
    operator fun contains(element: E): Boolean
    // truncated
}

interface MutableList<E> : List<E> {
    fun add(element: E): Boolean
    fun remove(element: E): Boolean
    fun clear()
    // truncated
}

class Foo(val someList: List<Int>) {
    init { // called during object construction
        assert(0 in someList)
    }
    fun doStuff() {
        // some stuff
        val idx = someList.indexOf(0) // implicit assumption:
                                       // init assert still holds!
        // some more stuff
    }
}

```

 **Figure 1** List and MutableList from Kotlin and client code.

In other words, since `MutableList` instances are also `List` instances, the best we can say is that `List` does not allow mutation and does not forbid mutation. An understandably-confused programmer may create an instance of the `Foo` class (line 17 of Figure 1) using a `MutableList`, which will be allowed by the type-checker. The list might at first satisfy the initial assertion, but the programmer may then clear it before calling `doStuff`. `doStuff` relies on the assertion in the constructor and dereferences a now-empty list, due to the mistaken assumption that `List` objects cannot change. Kotlin’s list hierarchy keeps us from taking advantage of the type checker to enforce our design decisions.

This shows how immutability-related violations of the LSP are particularly insidious. For this reason, the Scala standard collections library and the Guava libraries for Java fully separate their mutable collections from the immutable ones [5, 7, 21].

Object state: concrete vs. abstract. One possible solution is to “freeze” the memory: create a copy of an object that disallows mutation of all fields. This “freeze” could be shallow (as C++’s `const` would create) or deep (essentially an expensive clone). Such a shallow “freeze” operation exists in languages such as JavaScript [8] and Ruby [4]. Both approaches have significant disadvantages as mentioned, and are not widespread. Moreover, object fields are sometimes used for internal bookkeeping in ways that permits – and requires – to update their values in situations where the object’s content is not conceptually changed. An example of this can be seen in `ImmutableLookupList` (Figure 2), where the field `lookupCache` is used for memoizing calls to `indexOf`. While the class indeed mutates this field, it does so in a way that is non-observable to the user. In such cases, memory freeze is too strong, as it would disallow these updates. This requires the same kind of escape hatch that `mutable` provided for `const`, which yet again puts the burden on the programmer to decide which fields present part of the visible state. In some cases, the distinction is not even possible, because a field may produce a visible effect for some, but not all, of the ways in which it can be mutated.

22:4 Constrictor: Immutability as a Design Concept

```

class ImmutableList<E> : List<E> {
    private var lookupCache: CacheEntry<E, Int>?
    val backingArray: Array<E>
    override fun indexOf(elem: E): Int {
        if(this.cache != null && this.cache!!.first == elem)
            return this.cache!!.second

        var ret = -1
        for(i in backingArray.indices())
            if(backingArray[i] == elem) {
                ret = i
                break
            }

        this.cache = CacheEntry(elem, ret)
        return ret
    }
    // truncated
}

```

Figure 2 A class that mutates fields but not in an observable way.

For example, in the standard implementation of the union-find data structure [33], some mutations to the pointer structure may cause visible mutation while others are just different ways of expressing the same data.

The problem with `ImmutableLookupList` is actually a problem with considering `lookupCache` to be part of the state. It is, of course, part of the *concrete state* of an `ImmutableLookupList` object, i.e., it is part of the memory allocated for the object. However, let us consider how `ImmutableLookupList` looks to an external observer: `lookupCache` is used in the implementation of the method `indexOf`, and mutated by it, but this mutation is not observable – through `indexOf` or any other method of `ImmutableLookupList`. It is, in other words, an “implementation detail”, never exposed to any client code. It does not impact the *abstract state* of the object [54]. What we need, therefore, is immutability of the *abstract state* of the object.

1.1 Our approach: views and view immutability

In order to separate the abstract state from the fields pertaining to internal implementation, we define an object’s *view*: the set of methods that expose the abstract state to the rest of the system. The guarantee we want, then, is that if the view of an object is immutable, and this property is enforced down the inheritance tree, the immutable hierarchy can safely accommodate mutations of internal state. An enforcement mechanism less rigid than `const` or frozen objects can allow optimizations like memoization and caching, while disallowing the introduction of visible mutation into the hierarchy.

We define for each object two sets of methods, the set of immutable methods I , annotated by the programmer as `@immutable`, which are methods that do not mutate the abstract state of the object, and the set of view methods V , annotated as `@viewmethod`, whose return values define the object’s abstract state. In the common case, $V \subseteq I$, and so `@viewmethod` also indicates `@immutable` (this is not theoretically required, but conserves user effort). Marking the class as `@immutable` has the same effect as marking each of the class’s methods as `@immutable`, with one notable distinction: the class annotation is inherited, and applies to *all* methods of the inherited class, including new methods that were not inherited from its parent class.

We then define the notion of *view-immutability* with regards to the view V such that when calling any method from I , the object’s internal state may change, but the abstract state exposed by V does not. While checking this property is not tractable, we show a relaxed property that can be checked, that implies the stronger property under certain conditions.

The notion of view-immutability is meant to be checked in a modular way – there is no need to verify anything regarding the client code, only the data structures themselves. We expect that common data structures in libraries be annotated with `@immutable` as needed, and client code can use these data structures with the desired guarantees.

Our theory is flexible enough to support weaker notions of immutability, e.g., temporary mutability during an init phase [51], or temporary *immutability*, e.g., immutable references in the type system guaranteeing that referenced objects do not change, as in Rust [40].

We implement our approach in a linter/verifier for Python programs named CONSTRUCTOR. We translate each class to an SMT encoding using our translating compiler, Py2SMT, then check whether each of the methods in I are indeed non-mutating.

Lightweight verification. CONSTRUCTOR does not verify the code for correctness; rather, it checks for adherence to design decisions, which is an easier problem. However, it can still fail: CONSTRUCTOR’s analysis is bounded, and its reliance on SMT inherits the solver’s limitations. Even with these limitations, CONSTRUCTOR can still act as a contract-checker. This hinges on the fact that immutability violations are usually not bugs but rather unintended violations of conscious design decisions made by different programmers, and as such, they rarely hide from the programmer – or from CONSTRUCTOR. Empirically, the immutability property depends *mostly* on the program’s dataflow and not on complex relationships between values. Sometimes there are some correlations that need to be tracked, e.g., in Figure 2 the cache variable’s value is returned to client code, and so needs to be consistent with a real value/index in the list. When the SMT solver returns *unknown*, there are two options: if CONSTRUCTOR is run as a verifier, these unknowns will be treated as violations, whereas if it is run as a linter, only violations for which the solver has returned an answer will be displayed to the user.

We evaluate CONSTRUCTOR on 51 examples of immutability-related design violations. Our evaluation shows that CONSTRUCTOR is effective at catching a variety of prototypical design violations, and does so in seconds. We also explore CONSTRUCTOR with two real world case studies, one fixing a design problem in a collections module, and the other introducing memoization into an immutable design pattern. Moreover, we explore human errors that could be made when providing CONSTRUCTOR with annotations.

Contributions. The contributions of this paper are:

- A definition of *view immutability*, and a relaxed definition that can be statically checked.
- An SMT-based algorithm for checking view immutability.
- Py2SMT, a compiler that encodes Python functions for SMT solvers.
- CONSTRUCTOR, a verifier/linter that implements our algorithm for Python programs.
- An empirical evaluation of CONSTRUCTOR and detailed analysis of the results.²

2 Overview

CONSTRUCTOR is a linter/verifier for Python, so, from now on, the examples will be written in Python. The general concepts are identical and we will be using full type annotations.

We continue our running example that consists of a list library that includes the interface `LookupList` in Figure 3. This interface only contains methods that allow for the *inspection* of instances of its implementors. The programmer’s intent was that instances of `LookupList` should not be mutated (visibly) through their methods. Users of the library rely on this assumption, which until now was only enforced by comments and naming conventions.

² Our replication package is available as a DARTS artifact [36].

22:6 Constrictor: Immutability as a Design Concept

```
@immutable
class LookupList[E]:
    @viewmethod
    def __getitem__(self, idx: int) -> E:
        pass

    def index_of(self, element: E) -> int:
        pass

    @viewmethod
    def get_size(self) -> int:
        pass
```

Figure 3 A Python interface for a list class with an `index_of` method.

The programmer seeks to formalize this assumption: they add the `@immutable` annotation `LookupList`. Because `LookupList` functions as an interface, the substitution principle [39] dictates that the `@immutable` annotation should hold for inheriting classes as well.

Consider two implementors of `LookupList` (Figure 4). One of them, `UpdatingLookupList`, violates this assumption by adding methods that mutate the state in a visible way. The other, `MemoizingLookupList`, also mutates an object field, but does not change the abstract state of the object as observed through the `LookupList` interface: the field `cached` is used for memoization: storing `index_of`'s most recent input/output. Since both classes update data in object fields, the distinction between them is not a simple semantic check.

Our goal is for CONSTRICTOR to warn the user about the `@immutable` annotation's violation in `UpdatingLookupList`, and not generate a spurious warning for `MemoizingLookupList`.

Immutable abstract state. The sense in which we would like `LookupList` to be immutable is that the return values of “getter” methods, such as `__getitem__`, do not change after calling any of `LookupList`'s methods. In this sense, their *abstract* state is represented by their “observing” methods, whose return values should not change if we wish to consider `LookupList` an immutable interface.

We call the set of methods representing the abstract state the class's *view*: if two objects can be viewed differently through these methods, they definitely do not represent the same conceptual object. Notice that defining the view as just `__getitem__` and `get_size` would be equivalent to defining it to be all three methods of `LookupList`, because for any implementation upholding the class contract, two instances agreeing on the return values of `__getitem__` and `get_size` for all parameters would also agree on the return values of the other two methods.

The choice of view is akin to defining the abstract object: `index_of` only exposes the first instance of every value, and different lists that share the locations of duplicate elements – it does not matter which elements as long as they are duplicates – would be equivalent under a view made up of only `index_of`. Moreover, if `LookupList` had a `contains` method returning whether an element is in the list *somewhere*, then a view comprising only `contains` would essentially define the abstract object to be equivalent to a set.

It is therefore important to choose a view that represents the intended abstract state for the class. Modeling a list essentially means modeling a partial function mapping indices to elements, which can be achieved with one of the views above. Between equivalent views, choosing the smallest one will reduce the size of formulas generated by CONSTRICTOR, which will usually reduce the tool's run time.

When considering both implementations of `LookupList`, it appears as though both implementations cause state mutation by changing fields. However, one, `MemoizingLookupList`, realizes the contract and does not mutate the state *visibly*, while the other, `UpdatingLookupList`, mutates the state in a way that can be observed from outside the class.

```

class MemoizingLookupList[E](LookupList):
    cached: Pair[int, E]
    data: list[E]
    size: int

    def index_of(self, element: E) -> int:
        if self.cached.second == element:
            return self.cached.first
        for i in range(self.size):
            if self.data[i] == element:
                self.cached = Pair(i, element) # mutation!
                return i
        return -1
    # truncated

class UpdatingLookupList[E](LookupList):
    data: list[E]
    size: int

    def index_of(self, element: E) -> int:
        for i in range(self.size):
            if self.data[i] == element:
                return i
        return -1

    def add(self, element: E):
        self.data.append(element) # mutation!
        self.size += 1 # mutation!

    def remove(self, element: E):
        self.size -= 1 # mutation!
    # truncated

```

Figure 4 Two implementations of the list interface from Figure 3.

This motivates us to define *view-immutability*: a class is view-immutable if calling any of its methods on any instance with any parameters does not affect the return values of any method *in the class's view*. This definition allows `MemoizingLookupList` and rules out `UpdatingLookupList`.

2.1 Reasoning about view-immutability

In order to verify view-immutability, and know that our assumptions about the abstract state hold, we would need to prove a very strong property: for every state that an object can reach, and for every method m that we would like to show is immutable, the state of the object before and after calling m are *indistinguishable* for any trailing sequence of methods in the object's view. In other words, calling m (or not calling it) does not change the information returned from the object's view.

In other words, we would be considering two sequences of calls on object o :

$$\begin{aligned} & init(\vec{a}); m_1(); \dots; m_k(); \mathbf{m}(); m_{k+1}(); \dots; m_{k+n}() \\ & init(\vec{a}); m_1(); \dots; m_k(); \quad \quad m_{k+1}(); \dots; m_{k+n}() \end{aligned}$$

where throughout the sequence, if m_i is part of the class view, the return value of m_i is the same. The calls up to m_k constitute the object's initialization phase, which defines all the reachable object states. We assume that all methods are deterministic, so the values returned during initialization are trivially equal, and it remains to be checked for m_{k+1}, \dots, m_{k+n} . This task is hard to automate because it requires reasoning about unbounded sequences of method calls. At the very least, some user intervention would be needed, in the form of data-structure invariants or other guidance [12, 15, 28].

View abstraction. Our approach is inspired by successful notions from the field of model checking [20]. Instead of tracking sequences of method invocations, we establish an invariant that holds at every step; one “step” being a synchronous method application $m_i(\vec{a})$ on two object states σ_1, σ_2 . The invariant is derived from our notion of *view*: we assume that the methods in V represent the abstract state of the object. Therefore we would like to maintain the invariant that the two states are *view-equivalent* – that is, all the view methods always return equal values when invoked on σ_1 and σ_2 . We denote this by $\sigma_1 \equiv_V \sigma_2$.

To translate the problem to model checking, object *states* are modeled as valuations to the object’s fields (with a signature as defined by the respective class declaration). Methods are then represented as transitions between states. We denote the transition from σ to σ' using the method m as $\sigma \xrightarrow{m} \sigma'$. The problem is reduced to safety verification with the *relational* invariant $\sigma_1 \equiv_V \sigma_2$.

While this abstraction deliberately omits some internal information about the state, which may introduce spurious warnings, this modeling makes the problem amenable to well-established model-checking techniques based on SMT. We employ a Floyd-style approach: we construct the control-flow graph of each method and then trace all control paths up to some bound. Every program statement is associated with a first-order semantics, which are composed along each path to construct a path transition relation. The transition relation for the method is the disjunction over all of these paths. More details are given in Section 5.

2.2 Validation steps

This subsection walks through how CONSTRCTOR performs the check as explained, using our motivating example `MemoizingLookupList` to illustrate how CONSTRCTOR is able to show that this class satisfies the `@immutable` contract despite benign mutations caused by its methods.

The `LookupList` interface is annotated as `@immutable`, indicating all its methods should be non-mutating. The developer of `LookupList` additionally annotates the `__getitem__` and `get_size` methods as `@viewmethod`, defining the view of the object. The `@viewmethod` annotations are inherited by `MemoizingLookupList` along with the `@immutable` annotation on the class. Note that the inherited `@immutable` annotation on the class requires all of its methods to be non-mutating, including ones that are not inherited from `LookupList`.

This annotated code is the input to CONSTRCTOR. CONSTRCTOR first checks that the view of `MemoizingLookupList` is *faithful*, i.e., can represent the abstract state of the class. It then verifies that all methods marked `@immutable` do not affect the values of the view.

Step 1: Encoding to SMT. First, we convert each Python method m to an internal representation describing an approximation of the changes it makes to the object. We denote this the *transition relation* of the function and label it TR_m .

For example, in the transition relation of `UpdatingLookupList.add`, the assignment of `self.size` on line 28 of Figure 4 is expressed as $\sigma'[size] = \sigma[size] + 1$. The method’s transition relation is the composition of the transitions of all statements across all execution paths, in the standard manner.

CONSTRCTOR’s semantics component is called Py2SMT, and it operates at the method level by enumerating all execution paths up to a bound (this is used, for example, in loops such as the one in Figure 1), collecting path constraints and constructing the composed transition relation TR_m symbolically for each method m . As is usually the case with bounded model checking [16], the computed TR_m is an approximation.

Step 2: Agreement formula. The transition relations of the view methods are used to compute a set of predicates that check whether two object states are view-equivalent, i.e. *agree* on the return values of all methods $m \in V$ (with any arguments). These predicates are constructed by considering all possible program states at the end of each method, where the starting states are two given object states σ_1, σ_2 , checking whether the return value is equal in both. A program state – unlike an object state – also evaluates all the local variables and, in particular, the method’s return value, which we denote $\sigma[\text{returned}]$. We use \vec{a} to denote the method’s call arguments, which occur in TR_m as free variables.

$$\begin{aligned} \text{agree}_m(\sigma_1, \sigma_2) &\triangleq \forall \sigma'_1, \sigma'_2, \vec{a}. \\ \text{TR}_m[\vec{a}](\sigma_1, \sigma'_1) \wedge \text{TR}_m[\vec{a}](\sigma_2, \sigma'_2) &\rightarrow \sigma'_1[\text{returned}] = \sigma'_2[\text{returned}] \end{aligned}$$

View equivalence is expressed symbolically by conjoining over all view methods. In this example, there are two:

$$(\sigma_1 \equiv_V \sigma_2) \triangleq \text{agree_getitem_}(\sigma_1, \sigma_2) \wedge \text{agree_get_size}(\sigma_1, \sigma_2)$$

Step 3: View Fidelity. Using the transition relation for all methods and the agreement formula, we compose for each method m the formula for checking the fidelity of the view:

$$\forall \sigma_1, \sigma_2, \sigma'_1, \sigma'_2, \vec{a}. \quad \sigma_1 \equiv_V \sigma_2 \wedge \text{TR}_m[\vec{a}](\sigma_1, \sigma'_1) \wedge \text{TR}_m[\vec{a}](\sigma_2, \sigma'_2) \rightarrow \sigma'_1 \equiv_V \sigma'_2$$

If this formula is found valid for all methods of the class, it means two objects that are visibly indistinguishable remain visibly indistinguishable after any operation. The formula is valid for all four methods of `MemoizedLookupList`, so its view is faithful.

Step 4: View Immutability. Finally, CONSTRUCTOR uses both the transition relation and the view-equivalence relation to construct the immutability check formula for each `@immutable` method: for every object state, executing the checked method on it will not change the view. For `MemoizedLookupList.index_of`, this means:

$$\forall \sigma, \sigma', \text{idx}. \quad \text{TR}_{\text{index_of}}[\text{idx}](\sigma, \sigma') \rightarrow \sigma \equiv_V \sigma'$$

The formula for `index_of` is valid, and it can be validated by an SMT solver. This verifies that `index_of` is view-immutable over V . In contrast, if we try the same with, e.g., `UpdatingLookupList.add`:

$$\forall \sigma, \sigma', \text{el}. \quad \text{TR}_{\text{add}}[\text{el}](\sigma, \sigma') \rightarrow \sigma \equiv_V \sigma'$$

The formula for `add` is *not* valid, and the solver is able to produce a counterexample to this property. For example, if $\sigma = \{\text{data} \mapsto [], \text{size} \mapsto 0\}$, the TR is satisfied by $\sigma' = \{\text{data} \mapsto [\text{el}], \text{size} \mapsto 1\}$; but these are not view-equivalent. In particular, `get_size()` returns 0 for σ , but 1 for σ' .

3 Definitions

In this section, we define the necessary components for CONSTRUCTOR’s analysis. Let C be a class with fields F and methods S .

► **Definition 1** (Object State). *The object state of an instance of C is its logical representation: an assignment giving a value for each field in F .*

22:10 Constrictor: Immutability as a Design Concept

► **Definition 2 (View).** A view of C is a set of methods $V \subseteq S$ that describe the abstract state of the class.

The view will usually contain getters for core fields of the class, while omitting memoization fields, caches and any other data that is not part of the object's abstract state. While there are usually many options for selecting V , any specific choice is an expression of intent.

► **Definition 3 (Method Term).** A method term τ for method $m \in S$ is an expression $m(p_1, \dots, p_k)$ where $p_{1..k}$ are concrete values of the corresponding parameter types. We denote $T(X)$ for $X \subseteq S$ to be the set of method terms for all $m \in X$. We use the shorthand $T \triangleq T(S)$

A method term τ , when operating on an object state σ , has a return value $(\sigma.\tau)$ and a post-state σ' , for which we denote $\sigma \xrightarrow{\tau} \sigma'$.

What we actually want is to reason about two objects being *indistinguishable* in the sense that view methods, which are the representation of the abstract state of the object, cannot tell them apart. If two objects disagree on the values of view method terms, they are clearly not indistinguishable. However, it is possible the objects agree on the values of view method terms, but after applying some method, view methods of the resulting objects will disagree. This can happen for arbitrarily long sequences of methods, motivating the following definition:

► **Definition 4 (Observable Indistinguishability).** Two objects σ_1^0, σ_2^0 are observably indistinguishable (OI) ($\sigma_1 \stackrel{\bullet}{=} \sigma_2$) with respect to view V if for all method terms τ_1, \dots, τ_k , whenever:

$$\begin{aligned} \sigma_1^0 &\xrightarrow{\tau_1} \sigma_1^1 \xrightarrow{\tau_2} \cdots \xrightarrow{\tau_k} \sigma_1^k \\ \sigma_2^0 &\xrightarrow{\tau_1} \sigma_2^1 \xrightarrow{\tau_2} \cdots \xrightarrow{\tau_k} \sigma_2^k \end{aligned}$$

it is the case that σ_1^k, σ_2^k agree on the values of all view methods from V .

Now, view immutability just means that method calls leave objects observably indistinguishable from their previous state:

► **Definition 5 (View Immutability).** A method $m \in S$ is view-immutable with respect to the view V if:

$$\forall \tau \in T(\{m\}). \forall \sigma, \sigma' \in \Sigma. \sigma \xrightarrow{\tau} \sigma' \rightarrow \sigma \stackrel{\bullet}{=} \sigma'$$

A class C is view-immutable if all of its methods are view-immutable, including methods in classes that inherit from C .

This definition is hard to check because observable indistinguishability requires checking arbitrarily long sequences of method calls. However, since we expect the values of the view to reflect the full abstract state of the object, we can consider the following, weaker definition:

► **Definition 6 (View Equivalence).** Instances σ_1, σ_2 of class C are view-equivalent ($\sigma_1 \equiv_V \sigma_2$) if they agree on the values of all method terms of view methods:

$$\forall \tau \in T(V), (\sigma_1.\tau) = (\sigma_2.\tau)$$

For this to work, we expect views to be *faithful* in their representation of the abstract state of the class. A problem arises if there exist two view-equivalent states, and some method term from T , such that when applying the term on both states, the resulting states are no

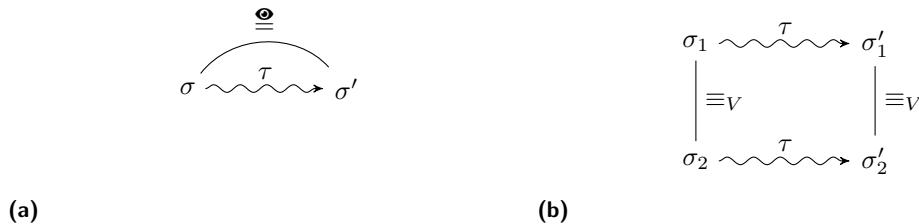


Figure 5 Illustrations of the definitions of (a) View Immutability (Definition 5) and (b) View Fidelity (Definition 7).

longer equivalent. Conceptually, this means that the view must be missing some information, because there exist two objects with the same view, but that are not interchangeable with respect to their subsequent behavior through application of class methods.

This motivates the following definition:

► **Definition 7 (View Fidelity).** *The view V is faithful (or exhibits view-fidelity) if for all two objects σ_1, σ_2 and for all method terms τ :*

$$(\sigma_1 \equiv_V \sigma_2 \wedge \sigma_1 \xrightarrow{\tau} \sigma'_1 \wedge \sigma_2 \xrightarrow{\tau} \sigma'_2) \rightarrow (\sigma'_1 \equiv_V \sigma'_2)$$

Actually, if the view is well-behaved (faithful), view equivalence between two objects implies the stronger property of observable indistinguishability.

► **Theorem 8 (Central Theorem).** *If V is a faithful view, and $\sigma_1 \equiv_V \sigma_2$, then $\sigma_1 \stackrel{\oplus}{\equiv} \sigma_2$.*

Proof. By induction on the length of the distinguishing method call sequence, and using view fidelity for the induction step. ◀

Our algorithm will rely on this theorem: we will check view fidelity and the preservation of view equivalence, and this will allow us to deduce observable indistinguishability.

4 Analysis

Our algorithm for checking if class C is view-immutable, shown in Algorithm 1, starts by computing the immutable set and the view set for the class, by using the class annotations:

- **@immutable**: A method labeled with **@immutable** must not affect the abstract state of the object; a class labeled as **@immutable** is a shorthand for labeling all methods as **@immutable** and all inheriting classes as **@immutable**.
- **@viewmethod**: adds a method to the view set of the object: the set of methods that, if they return the same values on two different objects, we consider them view-equivalent. A **@viewmethod** annotation also implicitly adds a **@immutable** annotation to the method. The user should aspire to providing the smallest view set.

These annotations are passed under inheritance.

In Algorithm 1, **IMMUTABLESET(C)** returns all methods annotated (directly or via inheritance) as **@immutable**, and **VIEWSET(C)** returns all methods annotated as **@viewmethod**.

For each method in the class, the transition relation is computed as a logical predicate between two SMT variable vectors with the appropriate method store signature. The method store signatures are a correspondence between names of memory locations used in methods and their types. In addition, each method store signature contains the special variable **returned** that represents the return value of the method. We denote this operation **GETTRMETHOD**, and it is implemented using Py2SMT, as explained in Section 5.

22:12 Constrictor: Immutability as a Design Concept

■ **Algorithm 1** Immutability checking algorithm.

```

procedure CHECKCLASS( $C$ )
Input: A class  $C$ 
Output: View unfaithful if the class view does not exhibit fidelity, and a mapping of methods
        to either Violation or No-violation otherwise.

 $V \leftarrow \text{VIEWSET}(C)$ 
 $\text{TRs} \leftarrow \{m \mapsto \text{GETTROFMETHOD}(C, m) \mid m \in C\}$ 
if not CHECKVIEWFAITHFUL( $V, \text{TRs}$ ) then
    return View unfaithful

Results  $\leftarrow \{\}$ 
for all  $m \in \text{IMMUTABLESET}(C)$  do
     $\Sigma = \text{METHODSTORESIGNATURE}(m)$   $\triangleright$  Collect types of fields and local variables
     $\varphi \leftarrow \forall \sigma, \sigma' : \Sigma. \text{TRs}[m](\sigma, \sigma') \rightarrow \text{AGREE}(V, \text{TRs})(\sigma, \sigma')$ 
    if CHECKSAT( $\neg \varphi$ ) then
        Results[ $m$ ] = Violation
    else
        Results[ $m$ ] = No-violation
return Results

```

■ **Algorithm 2** View equivalence checking algorithm.

```

function AGREE( $V, \text{TRs}$ )
Input: A set of view methods  $V$  and their transition relations
Output: The set's agree $_V$  predicate
 $\Sigma_s \leftarrow \{f \mapsto \text{METHODSTORESIGNATURE}(f) \mid f \in V\}$ 
return  $\lambda \sigma_0, \sigma_1. \bigwedge_{f \in V} (\forall \sigma'_1, \sigma'_2 : \Sigma_s[f]. (\text{TRs}[f](\sigma_1, \sigma'_1) \wedge \text{TRs}[f](\sigma_2, \sigma'_2)) \rightarrow$ 
 $\sigma'_1[\text{returned}] = \sigma'_2[\text{returned}])$ 

```

Next, the view fidelity of the full class is checked: the TRs are used to create a formula directly based on the definition of view fidelity, and its validity is checked. We denote this CHECKVIEWFAITHFUL in Algorithm 1. If the view is unfaithful, a meta-warning is issued.

Then, for each method in the immutable set I , the algorithm constructs a formula that searches for a counterexample to the immutability of the method. First, we compute a formula that is satisfied between two states that are view equivalent by using the AGREE(V, TRs) function, shown in Algorithm 2, on the set of view methods and their transition relations. Next, we use the result of AGREE to construct a formula that is satisfied by states that are not view-equivalent to their sequent states after application of the method. If the formula is satisfiable, then the class is mutable, and this method is a mutator.

This is essentially a reduction of the problem to model checking. Advancements in SMT solver technology can be applied to achieve better performance in our method as well (also see Section 6.6).

Strengthening optimization. One optimization that we found useful in our implementation is strengthening the claim and trying to prove $\text{TRs}[m](\sigma, \sigma') \rightarrow (\sigma = \sigma')$ instead of $\text{TRs}[m](\sigma, \sigma') \rightarrow (\sigma \equiv_V \sigma')$ in cases where the SMT solver returned `unknown`. This is a stronger property that is easier to check and holds in some cases. If that is the case, we can consider the method as a non-violation.

Correctness. The correctness of the algorithm relies on the following claim:

► **Theorem 9** (Algorithm Correctness). *If V is a faithful view, and for any method m of the class C :*

$$\forall \tau \in T(\{m\}). \forall \sigma, \sigma'. \sigma \xrightarrow{\tau} \sigma' \rightarrow \sigma \equiv_V \sigma'$$

then the class C is view-immutable w.r.t. the view V .

Proof. Let σ, σ' be states such that $\sigma \xrightarrow{m} \sigma'$. We can deduce that $\sigma \equiv_V \sigma'$. For view immutability, we need to prove that $\sigma \cong \sigma'$. We use Theorem 8 and the fidelity of the view V to deduce the desired property. ◀

5 Implementation

In this section we describe implementation details and design choices of CONSTRCTOR. Of these, the lion’s share is our compiler, Py2SMT.

Py2Smt. Py2SMT computes the overapproximations of transition relations of functions and the signatures of classes and functions for CONSTRCTOR. It is implemented using the Z3 [23] Python API.

Py2SMT creates a CFG for each Python function, and optimizes it in order to reduce graph size and path length. Function calls that have no summary SMT encoding are inlined into the graph, which means recursion is currently not supported. On the resulting graph, each path from the start vertex to the end vertex represents a potential execution path of the function. Py2SMT translates each operation to its SMT encoding, and all paths through the function are joined by a logical OR operation.

This translation depends on finite paths, so loops require special care: when the number of iterations is known at compile time, loops are completely unrolled. Unbounded loops, on the other hand, are unrolled and truncated to a configurable maximum length of program steps, rather than a fixed number of iterations – 100 steps in our evaluation – to create finite paths. This creates an underapproximation of the program’s behavior [16].

Moreover, since precise encoding of loops as logical formulas in decidable fragments of first-order logic is fundamentally impossible, Py2SMT currently supports `for` only in the cases of `range` iterations and iterations over lists. These are implemented by (i) utilizing the theory of sequences; and (ii) automatically converting `for` loops to a `while`-like form.

Py2SMT supports most built-in types: integers, floats, booleans, strings, lists, and dictionaries. It also supports arbitrary data types represented by classes, as well as generic classes using the bracket syntax from Python 3.12 [1]. Inheritance is also supported. Py2SMT relies on type hints for method signature inference in some cases. These can be provided by the user, or supplied by any type inference tool, such as PYTYPE [9].

Py2SMT supports reference types and treats class types in the same way as Python – all arguments are passed by reference, except for primitive types.

Use of solvers. Because the formulas created by CONSTRCTOR are at times large and complex, and SMT solvers may have different strengths, CONSTRCTOR first tries CVC5 [14] and, if it returns `unknown`, also tries Z3 [23].

CONSTRCTOR has two different operation modes, that differ in their behavior in the case that both solvers return `unknown` both for the original formula and for the heuristically strengthened one. In “linter-mode”, `unknown` is treated as “no-violation”, while in “verifier-mode”, `unknown` is treated as “violation”.

Unknown view fidelity. Algorithm 1 starts by attempting to prove view fidelity. CONSTRICTOR gives a meta-warning if it detects the view is not faithful, and proceeds if the view is faithful. If it cannot prove either (i.e., the solver returns `unknown`) it assumes the view is faithful and proceeds. This does not necessarily mean that the algorithm will result in an `unknown`, since the view fidelity check reasons about methods that the rest of the algorithm disregards.

6 Evaluation

Our evaluation is guided by these research questions:

- RQ1: Can CONSTRICCTOR validate a plethora of hierarchy-related design violations, as well as other cases involving immutability violations and non-violations?
- RQ2: Can CONSTRICCTOR validate realistic modules implementing data structures meant to be used in a larger projects?
- RQ3: What is the impact of certain types of annotation mistakes on CONSTRICCTOR?

6.1 Benchmarks

We collected 51 benchmarks comprising two sets:

- *Inheritance*: 24 examples of classes in four immutable class hierarchies, including both design violations and non-violations. Violations in this set include adding mutators to an immutable class, overriding immutable methods in a mutating way, defining a view of the object that returns part of the class’s internal state, etc. Some are classic examples of inheritance in object-oriented programming, and others are synthetic, created to measure CONSTRICCTOR’s performance for different sources of design-related immutability violations.
- *Non-inheritance*: 19 examples of immutability violations and non-violations in cases unrelated to immutable hierarchies. These exercise CONSTRICCTOR on a wider array of design issues, taken from online tutorials and the official language documentation for C++ [10]. Benchmarks originally in C++ were manually translated to Python and annotated such that every `const` C++ method is marked as `@viewmethod`.
- *Aspects & Limitations*: 8 synthetic benchmarks crafted to demonstrate various aspects and limitations of CONSTRICCTOR’s technique. The four types of benchmarks in this set explore: 1) loops are unrolled: violations hidden by the unrolling bound; 2) complicated view fidelity checks: views that are not trivially faithful, and the fact that checking view fidelity is separate from immutability violation checks, so the latter can succeed even when the former fails; 3) state space is overapproximated: one benchmark showing how unreachable code can cause a violation due to the overapproximation of object states; and 4) variable types must be explicitly specified: one benchmark showing cases where type inference cannot give an unambiguous answer without user-provided type annotations.

The *Inheritance* set contains 24 benchmarks, together measuring 778 lines of code (avg 32.4LOC) across 70 methods. The set contains 32 loops. Three methods suffer from intentional annotation mistakes. Six benchmarks contain lists and two contain dictionaries.

The *Non-inheritance* set contains 19 benchmarks, together measuring 806 lines of code (avg 39LOC) across 66 methods. The set contains 28 loops. One method suffers from intentional annotation mistakes. Eight benchmarks contain lists and three contain dictionaries.

The *Aspects & Limitations* set contains eight benchmarks, together measuring 248 lines of code and 20 methods.

Each `@immutable` and `@viewmethod` method is classified according whether its implementation violates the annotation. Our benchmark suite contains the following composition:

■ **Table 1** CONSTRICTOR results on the *Inheritance* benchmarks.

	Class	$ I $	$ V $	Fidelity exp/act	Violations Found	Success	Time (ms)
Lists	List	1	1	✓✓	0	✓	3107
	MutableList	2	1	✓✓	1	✓	2571
	AlrightPoint	3	2	✓✓	0	✓	366
	EvilPoint	4	2	✓✓	2	✓	301
	GoodPoint	3	3	✓✓	0	✓	267
	InauspiciousPoint	4	3	✓✓	1	✓	275
	MaliciousPoint	3	3	✓✓	1	✓	293
	MutablePoint	2	2	✓✓	0	✓	169
Points	Point	2	2	✓✓	0	✓	185
	WrongfullyAnnotatedMutablePoint	3	3	✓✓	1	✓	300
	EvilHashSet	1	1	✓✓	1	✓	3226
	GenericSet	1	1	✓✓	0	✓	47
	HashSet	1	1	✓✓	0	✓	10346
	MoveToFrontListSet	2	1	✓✓	2	✗	11268
	WrongImplMoveFrontListSet	2	1	✓✓	0	✗	2040
	ColoredShape	1	1	✓✓	0	✓	95
Shapes	EvilMemoizedRectangle	4	4	✓✓	1	✓	669
	EvilSquare	2	1	✓✓	1	✓	149
	LeakyMemoizedRectangle	4	4	✓✓	1	✓	876
	MemoizedRectangle	3	3	✓✓	0	✓	539
	Rectangle	4	4	✓✓	0	✓	656
	SimpleWrongImplRectangle	2	2	✓✓	1	✓	226
	SizedShape	1	1	✓✓	0	✓	94
	WrongfullyImplementedRectangle	4	4	✓✓	1	✓	841

Precision: 0.92 Recall: 0.92

exp: expected act: actual $|I|$ and $|V|$ include inherited annotations

	non-violations	violations	total classes
<i>Inheritance</i>	12	12	24
<i>Non-inheritance</i>	10	9	19
<i>Aspects & Limitations</i>	5	3	8

For all experiments, we define *precision* as the percentage of *no violation* detections made by the tool that were correct and *recall* as the percentage of actual non-violations that were correctly flagged as *no violation* by CONSTRICCTOR. All detections are at the function level.

All experiments ran on a 2022 MacBook Pro with an M2 processor and 16 GB of RAM.

6.2 RQ1: Design violations

To test RQ1, we ran CONSTRICCTOR on all three benchmark sets. CONSTRICCTOR ran on each benchmark separately, without caching the compilation results of PY2SMT. The timeout for CONSTRICCTOR was set at 10 minutes. We recorded the full runtime of CONSTRICCTOR for each class, the result of testing view fidelity, and the result of CONSTRICCTOR for each method in I .

The results for *Inheritance* are shown in Table 1 and *Non-inheritance* and *Aspects & Limitations* in Table 2. The aspect/limitation of each *Aspects & Limitations* benchmark is denoted by a superscript. Displayed times are an average over 10 runs. The repeated runs did not differ significantly, except for the `Graph` benchmark (marked with an asterisk in Table 2), which is discussed below. We computed the precision and recall of CONSTRICCTOR on the *Inheritance* and *Non-inheritance* benchmark sets: since many *Aspects & Limitations* benchmarks are designed to fail, including them does not make sense.

22:16 Constrictor: Immutability as a Design Concept

Table 2 CONSTRICTOR results on the *Non-inheritance* and *Aspects & Limitations* benchmarks.

Class	I	V	Fidelity exp/act	Violations Found	Success	Time (ms)
<i>Non-inheritance</i>	BiCounterFirst	2	1 ✓ ✓	0	✓	225
	BiCounterSecond	2	1 ✓ ✓	0	✓	304
	BinarySearchTree	2	1 ✓ ✓	1	✓	9459
	CachedList	1	1 ✓ ✓	1	✗	292
	CounterWithAccessCount	2	1 ✓ ✓	0	✓	225
	DefaultDict	2	1 ✓ ✓	0	✓	261
	EvilBinarySearchTree	2	2 ✓ ✓	2	✗ [†]	17346
	EvilUnionFind	1	1 ✓ ✓	1	✓	857
	Graph*	2	1 ✓ ✓	1	✓	2762
	ImmutablePerson	3	3 ✓ ✓	0	✓	205
	ImmutableRgb	3	2 ✓ ✓	1	✓	9823
	ListWithAccessCount	1	1 ✓ ✓	0	✓	12379
	MultiplyingDictionary	1	1 ✓ ✓	0	✓	6151
	MutablePerson	4	3 ✓ ✓	1	✓	381
	NumberShuffler	6	1 ✓ ✓	2	✓	477
	StringShuffler	2	1 ✓ ✓	0	✓	237
	UnionFind	1	1 ✓ ✓	0	✓	774
	WrongfullyAnnotatedCachedList	2	2 ✓ ✓	2	✓	403
	WrongfullyImplementedCollatz	2	1 ✓ ✓	1	✓	6823
Precision: 1.00 Recall: 0.90						
<i>Aspects & Limitations</i>	Collatz ¹	2	1 ✓ ✓	0	✓	5914
	FaithfulClass ²	1	1 ✓ ✓	0	✓	138
	FlaggedValue ²	1	1 ✗ ✗	0	✓	103
	LongLoopMutator ¹	2	2 ✓ ✓	0	✗	timeout
	UnreachablyMutating ³	2	2 ✓ ✓	1	✗	170
	VariableTypesMatter ⁴	4	2 ✓ ✓	1	✓	394
	ViewMutatingButFaithful ²	2	2 ✓ ✓	1	✓	160
	ViewNonMutatingButUnfaithful ²	2	1 ✗ ✗	0	✓	77

¹loops are unrolled ²complicated view fidelity checks ³state space is overapproximated

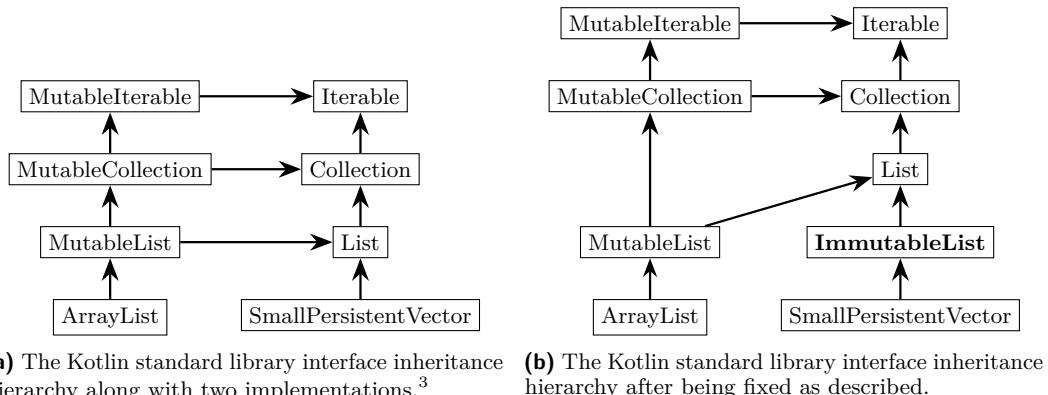
⁴variable types must be explicitly specified

exp: expected act: actual [†]unknown

CONSTRICCTOR checks all benchmarks but one in under 13 seconds, and all but 8 of the benchmarks (84.3%) in under 5 seconds. 46 benchmarks successfully flag all violations and find no spurious violations. One benchmark was marked as `unknown` by the SMT solver. CONSTRICCTOR succeeds in checking view fidelity for all benchmarks: all unfaithful views in Table 2 are accurately reported.

`Graph` is the only benchmark whose runtimes differed significantly across its 10 runs: two runs completed in under 3 seconds, two more runs completed in about 4 seconds, while the other six completed in about 12 seconds. This discrepancy is due to variations in solver run times; other components of the benchmark's run time did not change between runs.

Six of 51 benchmarks fail. Of these, 2 are *Aspects & Limitations* benchmarks designed to fail (of which, one times out), two benchmarks from the *Non-inheritance* set, and two benchmarks from the *Inheritance* set. Benchmarks `CachedList` (*Non-inheritance*) and `MoveToFrontListSet` (*Inheritance*) find a spurious violation by starting at an unreachable state of the object. Benchmark `WrongImplMoveFrontListSet` (*Inheritance*) misses a violation because the mutation occurs after the bound for loop unrolling. Benchmark `EvilBinarySearchTree` (*Non-inheritance*) was marked as `unknown` by the SMT solver.



■ **Figure 6** Inheritance hierarchies used in the first RQ2 case study.

We conclude that CONSTRUCTOR verifies designs that are view-immutable but do not pass simple C++-style const-checking, and finds design violations where mutation of the abstract state occurs.

6.3 RQ2 – Case Study 1: Kotlin lists

As our first case study, we consider the Kotlin standard library list hierarchy discussed in Section 1, with two implementing classes: the mutation-supporting `ArrayList` from the standard library, and the fully immutable `SmallPersistentVector` from the extension library `kotlinx.collections.immutable` [6]. Figure 6a summarizes the module’s initial hierarchy.

The library’s developer wants to annotate their code for CONSTRUCTOR. This involves:

1. Declaring for each class and interface a set of view methods,
2. Annotating some interface with `@immutable`, which will be inherited, and
3. Running CONSTRUCTOR on the classes in the hierarchy.

Technical setup. This case study is comprised of three copies of the eight classes in Figure 6a in three copies that are identical except for the location of the `@immutable` annotation, and a fourth version with the nine classes in Figure 6b. The interfaces were taken from the Kotlin standard library and translated verbatim, modeling abstract methods as empty methods (this makes no difference for CONSTRUCTOR).

Kotlin uses Java’s `ArrayList`, which we translated to Python as faithfully as possible: arrays were converted to lists which are used like arrays. Overloaded methods in Java were translated with different method names as Python does not support overloading. We attempted to model as many methods as possible: `trim_to_size`, `ensure_capacity`, `grow`, `get_size`, `is_empty`, `contains`, `index_of`, `last_index_of`, `to_array`, `get_element_data`, `get`, `set`, `add`, `remove`, `remove_at`, `_hash_`, `clear`, `add_all`, `remove_all`, `retain_all`, `iterator`, and `contains_all` are all modeled. Two subsets of its public methods were not modeled: (i) `listIterator`, `iterator`, `sublist`, `spliterator` because Py2SMT does not support internal classes, and (ii) `forEach`, `removeIf`, `sort`, `replaceAll` because Py2SMT does not support function objects. `SmallPersistentVector` was similarly translated as faithfully as possible, implementing `_presized_buffer_with`, `get_size`, `add`, `get`, `contains` and `index_of`.

³ Kotlin’s original hierarchy is taken from <https://kotlinlang.org/docs/collections-overview.html#collection-types>

 **Table 3** Run times for the case study in Section 6.3.

Class	Time (ms) for <code>@immutable</code> on			Result	Time (ms) for <code>@immutable</code> on	
	Iterable	Collection	List		ImmutableList	Result
<code>ArrayList</code>	11762	11902	12067	Viol.	5994	No viol.
<code>SmallPersistentVector</code>	2069	2443	2018	No viol.	2060	No viol.

Each of the three copies of the hierarchy in Figure 6a is about 290 lines of code overall. Specifically, our `ArrayList` is 150 lines of code compared to 511 lines of Java, excluding comments and internal classes. The hierarchy in Figure 6b is 296 lines of code and 51 methods overall. Times for all runs of CONSTRCTOR in this case study are shown in Table 3.

First attempt. The programmer declares `get_size` on `Collection` and `get` on `List` (which inherits the annotation on `get_size`) as view methods. They then try annotating `List` with `@immutable`. After running CONSTRCTOR on the classes in the hierarchy, `ArrayList` and `SmallPersistentVector`, CONSTRCTOR will issue a warning on `ArrayList`, which inherits `List`'s `@immutable` annotation but is mutable. CONSTRCTOR flags `ArrayList`'s `add` method as an immutability violation. Since `SmallPersistentVector` does uphold its inherited `@immutable` annotation, it is not flagged as a violation. The programmer then tries moving the `@immutable` annotation to either the `Iterable` or `Collection` interfaces, getting the same result.

In fact, the only class in Figure 6(a) on which the `@immutable` annotation would not cause a violation flag by CONSTRCTOR is `SmallPersistentVector`, on which it is useless. Overall, no interface in the hierarchy represents the immutability properties we expect, and CONSTRCTOR can detect this problem in the hierarchy.

The fix. To fix this issue, the programmer now separates the mutable and immutable hierarchies by creating a new interface: `ImmutableList`, which extends the `List` interface (as seen in Figure 6b). Now there is a clear separation between definitely-mutable classes and definitely-immutable classes. The programmer does not need to change the `@viewmethod` definitions to do so.

The programmer reruns CONSTRCTOR on the class hierarchy as previously described and gets no violation flags. This case study shows how CONSTRCTOR can help developers uphold immutable hierarchy constraints and declare them to their users.

6.4 RQ2 – Case Study 2: Red-Green trees

For our second case study, consider immutable trees with bidirectional references, i.e., both `children` and `parent` references. Smith [52] describes the problem: due to the immutability, we need to set the `parent` and `children` fields during initialization. However, initializing the tree with a `parent` field requires building it top-down, and initializing the tree with a `children` field requires building it bottom-up. These two requirements are contradictory.

Technical setup. We begin with a naïve implementation: a `Node` class with `parent`, `children`, and `data` fields. The class has `get_data`, `get_parent`, `get_children` and `add_child` as its methods, and is meant to be constructed top-down, setting the `parent` field upon construction. After construction, it is now possible to traverse the structure bottom-up and call the `add_child` method to initialize the `children` field.

We implemented this class in Python in 14 non-empty lines. We annotated the class as `@immutable` and annotated the `get_data`, `get_children` and `get_parent` methods as `@viewmethod`.

Table 4 Run times for CONSTRUCTOR on implementations of a bidirectional tree in Section 6.4.

Implementation	Run time (ms)	Result
Naive	322	Violation
Original Red-Green Tree	413	No violation
Memoized Red-Green Tree	582	No violation

As expected, CONSTRUCTOR returns a *violation* on this class, pointing out that `add_child` visibly mutates the class. The run time of CONSTRUCTOR can be found in Table 4.

First attempt. Red-Green trees [38] are a data structure used in the ROSLYN compiler for the .NET framework [3]. Red-Green trees solve the problem of bidirectional references by using two separate node objects to represent each tree node: an internal (and possibly mutable) *green* node and an immutable *red* node. The red tree serves as an immutable façade; the user never sees the green nodes. A green tree is constructed bottom-up, initializing each green node with its children. The red tree never exists as a tree, but rather red nodes are created on the fly to match each green node whenever the children of a red node are accessed. Since `get_children` is a computation instead of a getter, a red node can be initialized with just its parent and internal green node and remain entirely immutable.

We translated the version by Smith, converting 38 lines of C# code to 50 lines of Python code. We marked the `RedNode` class as `@immutable`, with `get_data`, `get_value`, `get_children` and `get_parent` as its view. As expected, CONSTRUCTOR did not detect a violation in this implementation since it stores nothing and mutates no field, even in a non-observable way.

However, this implementation is very inefficient, as it creates new red nodes representing the children of a given node in every call to `get_children`. This can cause both direct run time overhead, and indirect GC overhead caused by the allocation of many small objects, as noted by Lippert [38]. We would like to improve the performance of our implementation.

The fix. We now add memoization to our Red-Green tree: the result of the `get_children` method is stored when first called. Since red trees are immutable there is no reason to recompute this field. The new implementation now measures 55 lines. We then ran CONSTRUCTOR again: CONSTRUCTOR still did not report a violation, because the mutation of a field within `get_children` is non-visible, preserving view immutability.

This case study shows CONSTRUCTOR’s utility not in a class hierarchy but rather on validating the implementation of an immutable data structure. Unlike the previous case study, since Red-Green Trees are used an internal data structure, the `@immutable` annotation would serve the project developers to ensure no changes made to the red trees break their immutability. The classes from both case studies are part of our artifact [36].

6.5 RQ3: Impact of incorrect annotations

In the following small case studies, we set out to explore CONSTRUCTOR’s behavior in the presence of incorrect annotation by the user. We examined four types of annotation mistakes, relating to the `@immutable` and `@viewmethod` annotations: (i) incorrect specification of the class view, (ii) not marking all relevant methods as `@immutable`, (iii) marking a method as `@immutable` instead of `@viewmethod` and vice versa, and (iv) inheritance causing a non-faithful view. Technically, using the correct annotations is the user’s responsibility. We expect CONSTRUCTOR to behave under incorrect annotation as if the given annotations reflect the user intention. The purpose of this research question is to explore the results in cases that can be a little more error-prone.

22:20 Constrictor: Immutability as a Design Concept

```

class ListWithAccessCount[E]:
    arr: List[E]
    size: int
    access_count: int

    @viewmethod
    def get(self, idx: int):
        self.access_count += 1
        return self.arr[idx]

    @immutable
    def get_size(self):
        self.access_count += 1
        return self.size

    def get_access_count(self):
        self.access_count += 1
        return self.access_count

    def add(self, elem: E):
        self.access_count += 1
        self.size += 1
        if len(self.arr) == self.size:
            self.arr.append(elem)
        else:
            self.arr[self.size] = elem

    def remove_last(self):
        self.access_count += 1
        self.size -= 1
# truncated

```

 **Figure 7** A class with an incorrectly annotated view.

Incorrect annotation of the view. Precise view annotations are required to meet the criteria for Theorem 9. Consider the class `ListWithAccessCount` in Figure 7. The view annotations on this class may seem correct to a novice, but the view is too small. The behavior of `get` is undefined for `idx > self.get_size()`, so two `List` objects may *agree* on the values of `get` for all indices for which it is defined, while still not representing the same list, because they have different sizes. Also, CONSTRICITOR issues a fidelity meta-warning on the view in Figure 7.

The user can also incorrectly select a view for `ListWithAccessCount` that is too large: e.g., by adding `get_access_count` to the view. This is wrong for two reasons: (i) marking `get_access_count` as a view method exposes `self.access_count`, which means `get` is now considered to be mutating, and (ii) the `@viewmethod` annotation also denotes `get_access_count` itself as immutable, but it also mutates `access_count` before returning it. This means it cannot be both immutable and part of the view. Running CONSTRICITOR on `ListWithAccessCount` after denoting `get_access_count` as `@viewmethod` the class is flagged as a violation (time: 404ms). We recall that not all “public” methods are expected to be view methods, only those that define the abstract state of the object – the guarantees mentioned in Section 1 for view-immutability only require that the class is seen as immutable through the view, but return values for other methods may be affected.

Not marking a method or class as `@immutable`. In this case, CONSTRICITOR will simply not check the method or class. Because it only impacts *what* is checked, not the view that CONSTRICITOR uses, this does not affect CONSTRICITOR’s performance on other methods/classes.

```

class SettableList[E]:
    arr: List[E]

    @immutable
    def get(self, idx: int):
        return self.arr[idx]

    @viewmethod
    def get_size(self):
        return len(self.arr)

    @immutable
    def set(self, idx: int, elem: E):
        self.arr[idx] = elem

    @immutable
    def add(self, elem: E):
        self.arr.append(elem)
    # truncated

```

Figure 8 A class with `@viewmethod` and `@immutable` swapped.

```

class Collection[E]:
    @viewmethod
    def contains(self, elem: E) -> bool:
        pass
    # truncated

class MyCoolCollection[E](Collection):
    def remove_first(self, elem: E):
        n = self.get_size()
        for i in range(n):
            if self.get(i) == elem:
                self.remove_at(i)
    # truncated

```

Figure 9 An example where annotation inheritance may cause a view to become unfaithful.

Marking a method or class as `@immutable` instead of `@viewmethod` and vice versa. This is analogous to a view that is too large (using `@viewmethod` instead of `@immutable`) or too small (vice versa). For instance, consider the class `SettableList` in Figure 8, whose view is only the `get_size` method. The method `get` is marked as `@immutable` even though the user probably intended for it to be a part of the view. The result only partially captures the class's abstract state. In the current state, CONSTRUCTOR will not flag a violation for `set` that is marked as `@immutable`, because the size of the list does not change.

View fidelity under inheritance. The class `Collection` in Figure 9 represents a collection interface similar to Kotlin's. By itself, the class and its view, `contains`, have no issues. However, a programmer extending it may not be aware that adding methods to an inherited class may cause the view inherited from the parent to be unfaithful. When extended, `MyCoolCollection`'s view is the `contains` method inherited from `Collection`.

The programmer adds to `MyCoolCollection` the method `remove_first`, which removes one instance of an element given as a parameter to the method. This method causes the view of `MyCoolCollection` to be unfaithful, despite there being no change in the view set itself: two collections with the same distinct elements would agree on the return value of `contains` for all arguments, but after running `remove_first`, a collection with one instance of each element would become empty, while a collection with multiple instances of some elements would remain non-empty.

The solution in this case is to add a `get_element_multiplicity` method, which would make the view faithful again.

6.6 Discussion

Our results explore the bounds of CONSTRCTOR’s implementation. In this subsection we tie them back to the theoretical aspects of the technique.

Overapproximation and underapproximation. CONSTRCTOR can find spurious violations because we are overapproximating the TR in multiple ways, most importantly by considering all possible states of an object, including unreachable states. This means CONSTRCTOR can (and does) flag an illegal mutation or leaking of internal state in a benign method when the model found by the solver has an object in such a state.

CONSTRCTOR can also miss violations because of its handling of loops via unrolling. Since the loop is unrolled to a fixed, finite depth, it may be truncated too soon, making a real mutation invisible to CONSTRCTOR. Additional optimizations to CONSTRCTOR, particularly to Py2SMT, or improvements in the SMT solver could allow loops to be unrolled to a greater depth while preserving a reasonable run time. The introduction of loop invariants could help CONSTRCTOR find these violations, but annotating loops with invariants would be an unreasonable burden to the user. Integrating loop invariant inference tools [27] may be a reasonable compromise, but is outside the scope of this work.

View Fidelity. The correctness of Algorithm 1 fundamentally relies on the class being checked having a faithful view. CONSTRCTOR can try to return a result even when the view fidelity check fails, but this result is potentially incorrect. Moreover, view fidelity takes into account all class methods, not only those checked by CONSTRCTOR, causing its check to take a significant portion of CONSTRCTOR’s runtime. In large projects, it may be useful to manually check view fidelity and configure CONSTRCTOR to not check fidelity by itself.

View equivalence is a bisimulation. View fidelity essentially means that view equivalence forms a bisimulation between two traces representing the sequence of method calls on an object. Checking view immutability then means checking whether the view equivalence bisimulation holds between two traces that are identical except for a single point where they diverge: one trace performs a step and the other performs a no-op. Our algorithm for checking view immutability can then be seen as a special case of the symbolic model checking algorithm for checking bisimulation between the two traces, with view equivalence as the candidate bisimulation relation. Indeed, one modern algorithm for bisimulation checking for infinite state spaces is based on SMT [55]. This increases our confidence in the ability of our method to generalize, and implies that future improvements in bisimulation checking can also be applied to our technique.

Reliance on type hints. Some type hints are fundamental to CONSTRCTOR’s approach, and cannot be fully replaced with type inference. This is because some logical claims are valid in some theories and invalid in others. For example, the benchmark `VariableTypesMatter` from the *Aspects & Limitations* set contains two methods with the syntactically identical code segment `if self.some == a1 + a2: self.some = a2 + a1`, which is non-mutating if `a1` and `a2` are integers but mutating if they are strings, because integer addition is commutative and string concatenation is not. Type inference is performed in most cases where it is possible. However, as in the above example, the types of parameters cannot be precisely inferred, so type hints are required for function parameters and field types.

Py2Smt. Py2SMT is expressive, but it has two sets of limitations: (i) unimplemented Python language constructs, e.g., tuples, format strings, and list-, set-, and dictionary-comprehensions, and (ii) language constructs that are not symbolically expressible in SMT, e.g., general `for` loops and full polymorphism. We still support many common special cases, including iteration over lists and `range` objects, which we consider to be the most important cases for `for` loops. Additional work on Py2SMT can extend the scope of CONSTRICITOR.

Reliance on SMT solvers. Even when the formula Py2SMT encodes is accurate, there is no guarantee an SMT solver will be able to decide it. Some theories, e.g., arrays in cases where the domains and ranges are not disjoint, are simply undecidable. In Py2SMT, reference types are represented by using a heap “array”, which is why complex heap-based structures may yield formulas that return `unknown`. Performance on other theories may vary from solver to solver, which is why CONSTRICITOR tries both CVC5 and Z3. For example, certain formulas in the theory of sequences, which Py2SMT uses to encode lists, are not decidable by Z3 but can be decided correctly by CVC5. This affects performance on benchmarks involving lists.

Solvers are not only limited in the types they can represent, but also in the operations on those types. However, in our search for benchmarks we found that most design violations do not involve complex logic as part of the mutation. Therefore, despite the relatively limited expressiveness of SMT solvers, CONSTRICITOR can be useful in finding design violations.

Solvers are also not a great burden on the performance of CONSTRICITOR: across all benchmarks from all three sets, the wait for solver calls is on average 78ms, with the vast majority finishing in under 110ms. This is a small percentage of the runtime of many of the benchmarks, and of it, the majority of the time is spent in proving view fidelity, rather than on the main proof. The rest of CONSTRICITOR’s run time is spent on compilation, as well as other tasks (e.g., building the formulas). Only one benchmark (`WrongfullyImplementedCollatz` from the *Non-inheritance* set) causes a solver call that takes over 1 second (1.71 seconds). In general, no benchmark reaches the timeout set to the solver (3 seconds). The one timeout in Table 2 times out before the solver is called. Across all benchmarks in all benchmark sets, all solver calls take 13.06 seconds in total.

6.7 Threats to validity

The main threat to validity of this work is that complex, real-world code can be less straightforward to annotate. There may be more than one way to annotate a class, and deciding on its view can itself be a design decision. We attempt to mitigate this threat by introducing RQ3 to demonstrate the effect of using less precise annotations, as an inexperienced programmer might. There are still other ways in which a programmer can incorrectly annotate their code, and they may affect our results.

Moreover, in large, logic-heavy classes, proving view fidelity is more likely to fail because it needs to reason about all methods in the class, not only the `@immutable` ones. When the solvers return `unknown` on the fidelity formula, the result of CONSTRICITOR may be unsound, requiring user intervention. This may be unsustainable in a large project setting.

7 Related work

Alternate definitions of immutability. The type of immutability most discussed in the literature is *reference immutability* – non-mutation of an object’s fields through a specific reference [17, 29, 34, 54]. Mutation can also be allowed only in certain contexts [31, 45, 51].

This contrasts with *object immutability* [13], objects whose fields cannot be mutated via any reference. Object immutability requires more complex analyses to enforce [42]. Both definitions may or may not be transitive [46, 48, 49].

Potanin et al. define *abstract immutability* [19, Section 2.4] that permits “benevolent” side effects, but do not define what these effects can be or how this property is enforced. Eyolfson elaborates on this definition [26], roughly describing a desired solution which does not exist and is similar to view immutability.

Pure functions are functions that do not have any side effects, and only depend on their parameters. This is a very strong form of non-mutability, uncommon in OOP. A less strict form is defined by the JETBRAINS `@Contract(pure)` annotation, which indicates that a method does not “affect program state and change the semantics” (but can itself be affected by the state) [2]. Helm et al. [32] and Stewart et al. [53] unify different flavors of side-effect freedom by representing different definitions as a lattice.

Observational purity is a form of purity in which classes can keep and mutate state for their own use, but the mutated state may not leak out of the class. This similar notion to view immutability was introduced by Naumann et al. [43] for the purpose of formal specifications, as (observationally-) pure functions can be used in logical assertions. A method for checking observational purity was introduced in [12], and requires the user to manually supply invariants and specifications for all methods, which is sensible for settings in which writing specifications for all methods is common practice. This is not suitable for software engineering, because programmers typically do not write logical specifications for their classes. CONSTRCTOR implicitly defines an invariant by using view methods, which is slightly less expressive but very lightweight in terms of annotation burden.

Coblenz et al. have compiled a comprehensive classification of immutability types [21], which includes most systems mentioned in this section.

Tools. A well-known work on enforcing reference immutability is JAVARI [35, 41, 47, 54]. JAVARI’s type system distinguishes *unassignable variables* and *read-only references*. The former is more similar to Java’s `final` keyword, while the latter is introduced as part of the type system similar to C++. Another type system is introduced by Milanova [42] and allows distinguishing “maybe mutable” values from “definitely mutable” values, but makes no distinction between a variable and the value it stores, which may be a reference itself. Zibin et al. introduced a method to enforce object or reference immutability without changing Java’s grammar by using generic type parameters [56]. They allow excluding fields from the abstract state, much like C++’s `mutable` keyword. There is some work on automatic inference of immutability qualifiers. Eyolfson [26, Chapter 4] introduced IMMUTABILITY CHECK, which automatically infers `const` qualifiers. Eyolfson also introduced a system that automatically checks and sanitizes writes through `const` references [25].

Applications of immutability. Immutability can be part of the specification of a method [45]. Even if it is not necessarily part of the required semantics, it can be proven as a lemma in order to support analyses such as alias analysis [22] or flow analysis [50].

In concurrency, immutability is often proved as an auxiliary property to show *commutativity* of actions [18] (employing a similar SMT-based technique). This is because calling non-mutating operations in any order should result in the same results for each respective called method. Gordon et al. [30] pursue this in the context of reference immutability.

8 Conclusion

Objects whose values remain constant are desirable in software design. Current verification solutions are either too restrictive, barring all changes to the object and not just ones reflected in the object's abstract state, or too permissive, allowing mutations that can be observed. In this work, we presented a new approachcentering around the *view* of an object, which represents its abstract state, and whose values are expected to remain constant.

We introduced the new concept of *view-immutability* which expresses that the object's view does not change in a abstract sense. This solution is implemented as a linter/verifier, CONSTRICTOR, using an SMT-based method, which checks that method bodies adhere to denoted immutability constraints.

CONSTRICCTOR successfully detects a variety of design violations, with precision and recall both over 85%. We explored two large realistic case studies of data structures for which we found immutability to be useful, and CONSTRICCTOR is able to validate immutability or report violations. We also explore a set of smaller case studies for CONSTRICCTOR's behavior with imprecise annotations.

References

- 1 8. Compound statements – Python 3.12.1 documentation. https://docs.python.org/3/reference/compound_stmts.html#type-params. [Accessed 12-Jan-2024].
- 2 Contract (java8 17.0.0 API) – javadoc.io. <https://www.javadoc.io/doc/org.jetbrains/annotations/17.0.0/org/jetbrains/annotations/Contract.html>. [Accessed 27-Apr-2023].
- 3 dotnet/roslyn: The Roslyn .NET compiler provides C# and Visual Basic languages with rich code analysis APIs. <https://github.com/dotnet/roslyn>. [Accessed 14-Apr-2024].
- 4 freeze (Object) – APIDock. <https://apidock.com/ruby/Object/freeze>. [Accessed 14-Jan-2024].
- 5 ImmutableCollectionsExplained · google/guava wiki. URL: <https://github.com/google/guava/wiki/ImmutableCollectionsExplained>.
- 6 Kotlin/kotlinx.collections.immutable: immutable persistent collections for Kotlin. <https://github.com/Kotlin/kotlinx.collections.immutable>. [Accessed 13-Jan-2024].
- 7 Mutable and Immutable Collections | Collections | Scala Documentation. URL: <https://docs.scala-lang.org/overviews/collections-2.13/overview.html>.
- 8 Object.freeze – JavaScript | MDN. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze. [Accessed 14-Jan-2024].
- 9 pytype – google.github.io. <https://google.github.io/pytype/>. [Accessed 12-Apr-2023].
- 10 Standard C++ const correctness FAQ – isocpp.org. <https://isocpp.org/wiki/faq/const-correctness>. [Accessed 27-Apr-2023].
- 11 String.Intern(String) Method (System) | Microsoft Learn. URL: <https://learn.microsoft.com/en-us/dotnet/api/system.string.intern>.
- 12 Himanshu Arora, Raghavan Komondoor, and G. Ramalingam. Checking observational purity of procedures. In Reiner Hähnle and Wil M. P. van der Aalst, editors, *Fundamental Approaches to Software Engineering - 22nd International Conference, FASE 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 11424 of *Lecture Notes in Computer Science*, pages 228–243, Cham, 2019. Springer. doi:10.1007/978-3-030-16722-6_13.
- 13 Shay Artzi, Adam Kiezun, Jaime Quinonez, and Michael D. Ernst. Parameter reference immutability: formal definition, inference tool, and comparison. *Autom. Softw. Eng.*, 16(1):145–192, March 2009. doi:10.1007/s10515-008-0043-7.

22:26 Constrictor: Immutability as a Design Concept

- 14 Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022. doi:10.1007/978-3-030-99524-9_24.
- 15 Michael Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *J. Object Technol.*, 3(6):27–56, 2004. doi:10.5381/jot.2004.3.6.a2.
- 16 Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Adv. Comput.*, 58(99):117–148, 2003. doi:10.1016/S0065-2458(03)58003-2.
- 17 Adrian Birka and Michael D. Ernst. A practical type system and language for reference immutability. In John M. Vlissides and Douglas C. Schmidt, editors, *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, pages 35–49, Vancouver, BC, Canada, October 2004. ACM. doi:10.1145/1028976.1028980.
- 18 Adam Chen, Parisa Fathololumi, Eric Koskinen, and Jared Pincus. Veracity: declarative multicore programming with commutativity. *Proc. ACM Program. Lang.*, 6(OOPSLA2):1726–1756, October 2022. doi:10.1145/3563349.
- 19 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*. Springer, 2013. doi:10.1007/978-3-642-36946-9.
- 20 Edmund M. Clarke, Orna Grumberg, Daniel Kroening, Doron A. Peled, and Helmut Veith. *Model Checking, 2nd Edition*. Cyber Physical Systems Series. MIT Press, 2018. URL: <https://mitpress.mit.edu/books/model-checking-second-edition>.
- 21 Michael J. Coblenz, Joshua Sunshine, Jonathan Aldrich, Brad A. Myers, Sam Weber, and Forrest Shull. Exploring language support for immutability. In Laura K. Dillon, Willem Visser, and Laurie A. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016, ICSE '16*, pages 736–747, New York, NY, USA, 2016. ACM. doi:10.1145/2884781.2884798.
- 22 Cristina David and Wei-Ngan Chin. Immutable specifications for more concise and precise verification. 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, OOPSLA '11*, pages 359–374, New York, NY, USA, 2011. ACM. doi:10.1145/2048066.2048096.
- 23 Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, March 2008. doi:10.1007/978-3-540-78800-3_24.
- 24 José Javier Dolado, Mark Harman, Mari Carmen Otero, and Lin Hu. An empirical investigation of the influence of a type of side effects on program comprehension. *IEEE Trans. Software Eng.*, 29(7):665–670, 2003. doi:10.1109/TSE.2003.1214329.
- 25 Jon Eyolfson and Patrick Lam. C++ const and immutability: An empirical study of writes-through-const. 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 8:1–8:25. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ECOOP.2016.8.

- 26 Jonathan Eyolfson. *Enforcing Abstract Immutability*. PhD thesis, University of Waterloo, Ontario, Canada, 2018. URL: <https://hdl.handle.net/10012/13507>.
- 27 Carlo A. Furia, Bertrand Meyer, and Sergey Velder. Loop invariants: Analysis, classification, and examples. *ACM Comput. Surv.*, 46(3):34:1–34:51, January 2014. doi:10.1145/2506375.
- 28 Juan P. Galeotti, Nicolás Rosner, Carlos López Pombo, and Marcelo F. Frias. Analysis of invariants for efficient bounded verification. In Paolo Tonella and Alessandro Orso, editors, *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, ISSTA '10, pages 25–36, New York, NY, USA, 2010. ACM. doi:10.1145/1831708.1831712.
- 29 Paola Giannini, Marco Servetto, and Elena Zucca. Types for immutability and aliasing control. In Vittorio Bilò and Antonio Caruso, editors, *Proceedings of the 17th Italian Conference on Theoretical Computer Science, Lecce, Italy, September 7-9, 2016*, volume 1720 of *CEUR Workshop Proceedings*, pages 62–74. DEU, CEUR-WS.org, 2016. URL: <https://ceur-ws.org/Vol-1720/full15.pdf>.
- 30 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*, OOPSLA '12, pages 21–40, New York, NY, USA, 2012. ACM. doi:10.1145/2384616.2384619.
- 31 Christian Haack and Erik Poll. Type-based object immutability with flexible initialization. In Sophia Drossopoulou, editor, *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, volume 5653 of *Lecture Notes in Computer Science*, pages 520–545. Springer, Springer, 2009. doi:10.1007/978-3-642-03013-0_24.
- 32 Dominik Helm, Florian Kübler, Michael Eichberg, Michael Reif, and Mira Mezini. A unified lattice model and framework for purity analyses. In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 340–350. ACM, 2018. doi:10.1145/3238147.3238226.
- 33 John E. Hopcroft and Jeffrey D. Ullman. Set merging algorithms. *SIAM J. Comput.*, 2(4):294–303, 1973. doi:10.1137/0202024.
- 34 Wei Huang, Ana L. Milanova, Werner Dietl, and Michael D. Ernst. Reim & reiminfer: checking and inference of reference immutability and method purity. 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*, OOPSLA '12, pages 879–896, New York, NY, USA, 2012. ACM. doi:10.1145/2384616.2384680.
- 35 Telmo Luis Correa Jr., Jaime Quinonez, and Michael D. Ernst. Tools for enforcing and inferring reference immutability in java. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, OOPSLA '07, pages 866–867, New York, NY, USA, 2007. ACM. doi:10.1145/1297846.1297929.
- 36 Elad Kinsbruner, Shachar Itzhaky, and Hila Peleg. Constrictor: Immutability as a Design Concept (Artifact). *Dagstuhl Artifacts Series*, 10(2), 2024. doi:10.4230/DARTS.10.2.9.
- 37 Zach Klippenstein. Two mutables don't make a right. <https://dev.to/zachklipp/two-mutables-dont-make-a-right-2kgp>, 2021. [Accessed 08-Jan-2024].
- 38 Eric Lippert. Persistence, façades and Roslyn's red-green trees | Fabulous adventures in coding. <https://ericlippert.com/2012/06/08/red-green-trees/>. [Accessed 14-Apr-2024].
- 39 Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, November 1994. doi:10.1145/197320.197383.

22:28 Constrictor: Immutability as a Design Concept

- 40 Nicholas D. Matsakis and Felix S. Klock II. The rust language. In Michael B. 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*, volume 34(3), pages 103–104. ACM, 2014. doi:10.1145/2663171.2663188.
- 41 Matt McCutchen and Dr. Michael Ernst. Putting Javari into Practice, 2006. URL: <https://api.semanticscholar.org/CorpusID:24269793>.
- 42 Ana L. Milanova. Definite reference mutability. 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 25:1–25:30, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2018.25.
- 43 David A. Naumann. Observational purity and encapsulation. *Theor. Comput. Sci.*, 376(3):205–224, 2007. Fundamental Aspects of Software Engineering. doi:10.1016/j.tcs.2007.02.004.
- 44 Stephen Nelson, David J. Pearce, and James Noble. Understanding the impact of collection contracts on design. 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 61–78. Springer, Springer, 2010. doi:10.1007/978-3-642-13953-6_4.
- 45 Igor Pechtchanski and Vivek Sarkar. Immutability specification and its applications. In José E. Moreira, Geoffrey C. Fox, and Vladimir Getov, editors, *Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande 2002, Seattle, Washington, USA, November 3-5, 2002*, JGI '02, pages 202–211, New York, NY, USA, 2002. ACM. doi:10.1145/583810.583833.
- 46 Sara Porat, Marina Biberstein, Larry Koved, and Bilha Mendelson. Automatic detection of immutable fields in java. In Stephen A. MacKay and J. Howard Johnson, editors, *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative Research, November 13-16, 2000, Mississauga, Ontario, Canada*, CASCON '00, page 10. IBM, 2000. URL: <https://dl.acm.org/citation.cfm?id=782044>.
- 47 Jaime Quinonez, Matthew S. Tschantz, and Michael D. Ernst. Inference of reference immutability. In Jan Vitek, editor, *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*, volume 5142 of *Lecture Notes in Computer Science*, pages 616–641, Berlin, Heidelberg, 2008. Springer. doi:10.1007/978-3-540-70592-5_26.
- 48 Tobias Roth, Dominik Helm, Michael Reif, and Mira Mezini. Cifi: Versatile analysis of class and field immutability. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*, pages 979–990. IEEE, 2021. doi:10.1109/ASE51524.2021.9678903.
- 49 Atanas Rountev. Precise identification of side-effect-free methods in java. In *20th International Conference on Software Maintenance (ICSM 2004), 11-17 September 2004, Chicago, IL, USA*, pages 82–91. IEEE Computer Society, 2004. doi:10.1109/ICSM.2004.1357793.
- 50 Tobias Runge, Marco Servetto, Alex Potanin, and Ina Schaefer. Immutability and encapsulation for sound OO information flow control. *ACM Trans. Program. Lang. Syst.*, 45(1):3:1–3:35, 2023. doi:10.1145/3573270.
- 51 Marco Servetto, Julian Mackay, Alex Potanin, and James Noble. The billion-dollar fix - safe modular circular initialisation with placeholders and placeholder types. In Giuseppe Castagna, editor, *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*, volume 7920 of *Lecture Notes in Computer Science*, pages 205–229, Berlin, Heidelberg, 2013. Springer. doi:10.1007/978-3-642-39038-8_9.
- 52 Yaakov Smith. Red-Green Trees. <https://blog.yaakov.online/red-green-trees/>. [Accessed 14-Apr-2024].
- 53 Arran Stewart, Rachel Cardell-Oliver, and Rowan Davies. Fine-grained classification of side-effect free methods in real-world java code and applications to software security. In *Proceedings of the Australasian Computer Science Week Multiconference, Canberra, Australia, February 2-5, 2016*, ACSW '16, page 37, New York, NY, USA, 2016. ACM. doi:10.1145/2843043.2843354.

- 54 Matthew S. Tschantz and Michael D. Ernst. Javari: adding reference immutability to java. In Ralph E. Johnson and Richard P. Gabriel, editors, *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. ACM, 2005. doi:10.1145/1094811.1094828.
- 55 Yunfan Zhang, Ruidong Zhu, Yingfei Xiong, and Tao Xie. Efficient synthesis of method call sequences for test generation and bounded verification. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, ASE '22, pages 38:1–38:12, New York, NY, USA, 2022. ACM. doi:10.1145/3551349.3556951.
- 56 Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kiezun, and Michael D. Ernst. Object and reference immutability using java generics. 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*, ESEC-FSE '07, pages 75–84, New York, NY, USA, 2007. ACM. doi:10.1145/1287624.1287637.

InferType: A Compiler Toolkit for Implementing Efficient Constraint-Based Type Inference

Senxi Li 

The University of Tokyo, Japan

Tetsuro Yamazaki 

The University of Tokyo, Japan

Shigeru Chiba 

The University of Tokyo, Japan

Abstract

Supporting automatic type inference is in demand in modern language development. It is a challenging task but without appropriate supporting toolkits. This paper presents InferType, a Java library that helps implement constraint-based type inference. A compiler writer uses InferType's classes and methods to describe type constraints and typing rules for type inference. InferType then performs constraint solving by translation to the Z3 SMT solver. InferType is equipped with our developed optimization technique. It reduces the search space for type variables by pre-computing the structures of those type variables for mitigating the performance bottleneck of constraint solving with deeply nested types. We use InferType to implement type inference for a subset of Python, and conduct experiments to evaluate how the developed optimization technique can affect the performance of type inference. Our results show that InferType's optimization can greatly mitigate the performance bottleneck for programs with deeply nested types, and can potentially improve the performance for large nested types.

2012 ACM Subject Classification Software and its engineering → Domain specific languages; Theory of computation → Type theory

Keywords and phrases Domain Specific Languages, Compilation, Static Analysis, Type Inference, Constraint Solving, SMT Solver

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.23

Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact)*:

<https://doi.org/10.4230/DARTS.10.2.11>

Funding This work was supported by JSPS KAKENHI Grant Numbers JP20H00578 and JP24H00688.

1 Introduction

The goal of this work is to provide an assisting tool for implementing compilers supporting type inference, which is an in-demand task in modern language development but not well supported by existing approaches. Several supporting toolkits for language development have been proposed such as lexer and parser generators [22, 41], and type checker generators by language workbenches [45, 18]. Since many modern languages support type inference that automatically reconstruct types for programs without explicit type annotations, compiler writers have to implement type inference when developing their own languages. Unfortunately, there are no applicable toolkits for supporting this labor-intensive and also challenging task.

Major static languages support type inference that can be performed in a simple bottom-up manner. It does not need a complex inference engine. However, in recent languages like TypeScript, more aggressive type inference is supported. For example, they may allow programmers to omit a type annotation for a function's return type. A program in such

 © Senxi Li, Tetsuro Yamazaki, and Shigeru Chiba;
licensed under Creative Commons License CC-BY 4.0
38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 23; pp. 23:1–23:28



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

23:2 InferType

languages may require complicated type inference when it includes mutually recursive functions. Its compiler may contain an intricate type inference engine that solves type constraints retrieved from a program.

One of the approaches to implementing such intricate type inference is utilizing modern Satisfiability Modulo Theories (SMT) solvers. Several static analyses including automatic type inference have been established based on SMT solving [4, 42, 44, 14]. However, it is not simple to implement an efficient type inference engine for practical usage even when using a powerful SMT solver. For example, we observe that the performance of type inference using a SMT solver may drastically degrade when the source program contains nested types such as function types that take other function types as arguments and list types that take other list types as arguments. Such types are frequently used in real-world programs for describing higher-order functions and data structures in hierarchy.

In this paper, we propose InferType, a domain-specific language embedded in Java for implementing constraint-based type inference. A compiler writer implements a constraint-based type inference engine for a target language by describing type constraints and encoding typing rules using InferType’s classes and methods. She then calls an InferType’s method to get a type inference result, which is a binding of type variables to concrete types that satisfies the described type constraints and encoded typing rules. InferType uses the Z3 [7] SMT solver to perform constraint solving for computing that binding by translating the described type constraints and typing rules to Z3 formulae.

We develop an optimization technique for InferType to mitigate the performance bottleneck for handling programs containing deeply nested types. InferType’s optimization is based on the idea of reducing the search space of type variables in SMT solving. It first pre-computes a structure of each type variable in the given type constraints described by the compiler writer. It then generates extra Z3 formulae that explicitly list the possible types each type variable ranges over based on the pre-computed structure for reducing the search space.

We use InferType to implement type inference engines for a small demonstrative language and a subset of Python. This Python subset does not allow explicit type annotations as type annotations are ignored by the ordinary Python without an external type checker. We choose this Python subset for evaluation since type inference for this subset requires complex typing rules to reconstruct static types in dynamically typed programs. We conduct experiments over a collected dataset to evaluate the performance of type inference by our implemented type inference engine for the Python subset, and also to validate the effectiveness of our developed optimization technique. Compared with an existing, manual type inference engine for Python using Z3, the implemented type inference engine using InferType is able to infer types for real-world programs with compatible performance. Furthermore, the experiment results show that the implemented type inference engine can handle programs with deeply nested types much more efficiently. We also observe that InferType’s optimization can potentially improve the performance of type inference for programs containing nested types. Our findings support the claim that InferType is pragmatically applicable of helping implement constraint-based type inference in language development.

The rest of this paper is organized as follows. Section 2 motivates the reader by a scenario of developing a small demonstrative language. Section 3 presents our proposal. Section 4 gives our experiment results by the implemented type inference engine using InferType. Section 5 relates our work to preceding researches and a brief conclusion ends this paper.

2 Implementing Type Inference in Language Development

Many modern languages have support for automatic type inference. Language developers, or compiler writers have to implement type inference when designing a new language. However, it is a challenging task without appropriate tool supports. Classical supporting tools for language development include lex, yacc [22] and bison [41], which are lexer and parser generators. There are also tool suites commonly called “language workbench”, which are development environments for making domain-specific languages [45, 11]. The Spoofax [18] language workbench, for instance, supports code generators that produce type checkers and editor plugins from high-level language definitions. However, existing tools do not support code generators for type inference.

For instance, let us consider implementing a type inference engine for a small language called *Mini-λ*. It automatically reconstructs types for programs without explicit type annotations. We use this language for demonstrative purposes because of its simplicity. One of the common approaches to implementing such a type inference engine is to use a constraint-based type inference. A number of existing type inference systems are formulated into a constraint-based approach because of its extensibility and flexibility [31, 35, 39, 38, 19]. We below consider implementing a constraint-based type inference for *Mini-λ*.

The syntax of expressions and types of *Mini-λ* is given in Listing 1.

```
e := n | r | s | x | fun x. e | e(e) | e + e
t := int | float | str | arrow<t, t>
```

Listing 1 Syntax of *Mini-λ* expressions and types

n, *r* and *s* range over integer, real numbers and string literals. *x* ranges over variables. *fun x. e* represents a function with parameter *x* and a body *e*. *Mini-λ* does not have explicit type annotations. Literals can be regarded as type-annotated expressions. Functions or function parameters do not have type annotations. *e(e)* represents a function application. *e + e* represents an addition of two sub-expressions. In *Mini-λ*, the plus operator can either add numbers or concatenate strings. The type syntax of *Mini-λ* consists of primitive types **int**, **float** and **str**, and the composite type **arrow** for describing function types.

Some of the typing rules in *Mini-λ*’s type system are given in Figure 1. **T-App** specifies how a function application is typed so that the applied expression must be an arrow type consistent with the argument type. The type constraint in the premises specifies the subtype relation that the type variables must satisfy. **T-Plus-Float** and **T-Plus-Str** specify how a plus expression is typed by overloading. The operand and resulting types should all be consistent with either **float** or **str**. **ST-Int-Float** specifies that **int** is a subtype of **float**. **ST-Arrow** specifies that two arrow types are subtype of one another if the parameter type is contravariant and the return type is covariant.

An example program in *Mini-λ* is given in Listing 2. We will use this example program for demonstration in the rest of the paper because of its simplicity although this program is simple enough to be able to perform type inference in a bottom-up manner.

```
fun f. fun x. f(x)
  (fun y. y + "a")
```

Listing 2 An example program in *Mini-λ*

This program first defines a function with parameter **f**; its body is another function with parameter **x**. The body of the function with parameter **x** is a function application **f(x)**. After that, the function with parameter **f** is applied to a function with parameter **y**. The resulting value of the whole expression is a function taking one argument, and its body concatenates the passed argument to a string literal “**a**”.

$$\begin{array}{c}
 \frac{\Gamma \vdash e_1: t_1 \quad \Gamma \vdash e_2: t_2 \quad t_1 <: \text{arrow}\langle t_2, t_3 \rangle}{\Gamma \vdash e_1(e_2): t_3} (\text{T-App}) \\[10pt]
 \frac{\Gamma \vdash e_1: \text{float} \quad \Gamma \vdash e_2: \text{float}}{\Gamma \vdash e_1 + e_2: \text{float}} (\text{T-Plus-Float}) \\[10pt]
 \frac{\Gamma \vdash e_1: \text{str} \quad \Gamma \vdash e_2: \text{str}}{\Gamma \vdash e_1 + e_2: \text{str}} (\text{T-Plus-Str}) \\[10pt]
 \frac{}{\text{int} <: \text{float}} (\text{ST-Int-Float}) \\[10pt]
 \frac{t_{21} <: t_{11} \quad t_{12} <: t_{22}}{\text{arrow}\langle t_{11}, t_{12} \rangle <: \text{arrow}\langle t_{21}, t_{22} \rangle} (\text{ST-Arrow})
 \end{array}$$

Figure 1 Part of the typing rules of Mini- λ .

To implement a constraint-based type inference for Mini- λ , a compiler writer first assigns a type variable to every variable and expression in a program. Then the compiler writer generates the relations between type variables by applying the typing rules while traversing the abstract syntax tree (AST) of the program. A relation between type variables, which we call a *type constraint*, is a logical assertion that describes how the type variables should be constrained. In this paper, we consider those type constraints in Listing 3.

```

constraint := t <: t
             | constraint ∧ constraint
             | constraint ∨ constraint
             | constraint → constraint
             | ¬ constraint
  
```

Listing 3 Type constraints

$<:$ represents the subtype relation between two types. \wedge and \vee represent logical conjunction and disjunction; \rightarrow and \neg represent logical implication and negation.

Type constraints are generated during AST traversal by applying part of the typing rules in the target language, which we call *syntax-related typing rules*. A syntax-related typing rule is a typing rule that includes program syntax symbols. For example, T-App, T-Plus-Float and T-Plus-Str are syntax-related typing rules in Mini- λ . The compiler writer will generate the type constraints in Listing 4 for the program in Listing 2 by traversing its AST.

```

(1) tf <: arrow<tx, tfx>
(2) arrow<tf, arrow<tx, tfx>> <: arrow<arrow<ty, tplus>, te>
(3) (ty <: float ∧ str <: float ∧ float <: tplus)
    ∨ (ty <: str ∧ str <: str ∧ str <: tplus)
  
```

Listing 4 Type constraints for the example program

$t_?$ represents the unique type variables assigned to the variables and expressions in the program. t_f , t_x and t_y are assigned to parameter f , x and y of the defined functions. t_{fx} and t_{plus} are assigned to the resulting types of $f(x)$ and $y + "a"$. t_e is assigned to the resulting type of the whole expression. Constraint (1) and (2) arise from T-App. For example, in (1), t_f is constrained to be a subtype of $\text{arrow}\langle t_x, t_{fx} \rangle$ generated from the application $f(x)$. Constraint (3) arises from T-Plus-Float and T-Plus-Str. This constraint must be properly included in the generated set of type constraints to represent the overloading of

the plus operator. It describes that the operand types, t_y and str (type of the string literal "a"), and the resulting type t_{plus} must be subtype and supertype of either `float` or `str`, represented using logical connectives.

After constraint generation, the compiler writer must then implement a constraint solver considering the rest of the typing rules to solve the generated type constraints. The solver must find a consistent binding of concrete types assigned to the type variables that satisfies all the type constraints. The solver must consider the other typing rules from the applied typing rules in AST traversal, which we call *subtype-related typing rules*, to find such a binding. Unlike syntax-related typing rules applied during AST traversal, a subtype-related typing rule derives a conclusion in the form of a subtype relation and a subtype-related typing rule does not include syntax symbols of program expressions. For example, the implementing solver for Mini- λ must consider `ST-Int-Float` and `ST-Arrow` in Figure 1 to solve the type constraints in Listing 4. Although well-known algorithms such as unification [37] and closure computation [33] have been developed, implementing these algorithms is technically labor-intensive and error-prone. Furthermore, implementing an efficient solver becomes more sophisticated especially for handling complex programs when developing real-world languages. As we will show in 3.3, it would be time consuming for inferring types of programs with nested types by a straightforward implementation of such a solver. Technical approaches are needed to overcome those challenges considering practical usage.

3 InferType

InferType is an embedded domain-specific language that helps compiler writers implement efficient constraint-based type inference. A compiler writer gives type constraints derived by syntax-related typing rules and subtype-related typing rules to InferType by using InferType's classes and methods. InferType then solves the given type constraints based on the subtype-related typing rules by invoking the Z3 SMT solver. To perform constraint solving efficiently, InferType pre-computes structures for type variables involved in the given type constraints, and then reduces the search space of the involved type variables in SMT solving.

InferType supports constraint-based type inference whose type system is expressed by one single binary type relation, which is often called "subtype". InferType assumes that the supported binary type relations hold reflexivity and transitivity. InferType does not support type systems expressed by more than one type relation. For example, InferType could not support a gradual type system [40] which contains a subtype relation and also a type consistency relation.

Below in this section, we first demonstrate how compiler writers can use InferType to implement their type inference engines in Section 3.1. Then we show how InferType performs type inference by translation to Z3 in Section 3.2. In Section 3.3, we present the pre-process of InferType for mitigating the performance bottleneck in constraint solving, which is our main scientific contribution in this paper.

3.1 Programming Interface

InferType is a Java library for implementing constraint-based type inference engines, which are softwares that automatically infer types for programs without explicit type annotations in the target languages. A compiler writer uses InferType's classes and methods to describe type constraints derived by syntax-related typing rules and encode subtype-related typing rules. Constraint solving is then performed by calling an InferType's method to get a type inference result.

3.1.1 Describing types and type constraints

First, types in the target language must be described in InferType. In InferType, a type is an object taking one string argument as the name of that type, and zero or more type arguments.

```
type(typename, type, type, ...)
```

For instance, a primitive type `int` in Mini-λ is described as

```
InferType inf = new InferType();
Type intTy = inf.type("int");
```

`inf` is an instance of InferType's main class. The compiler writer declares this instance to describe types, type constraints and encode subtype-related typing rules. `Type` is the class representing types. Primitive types are described as types taking zero type argument in InferType. A composite type `arrow<int, str>` is described as

```
inf.type("arrow", inf.type("int"), inf.type("str"))
```

InferType also deals with type variables. A type variable is an object taking one string identifier.

```
typevar(identifier)
```

A type variable t_k is described as

```
inf.typevar("t_k")
```

Type variables can also be used as type arguments to make a composite type. An arrow type `arrow< t_k , int>` is described as

```
inf.type("arrow", inf.typevar("t_k"), inf.type("int"))
```

InferType can also help produce a type variable with a generated, unique string identifier by calling `inf.typevar` without an argument.

```
inf.typevar() // a fresh generated type variable
```

Then, the compiler writer describes type constraints generated by applying syntax-related typing rules during AST traversal, and gives the type constraints to InferType for type inference. A compiler writer describes the type constraints using InferType's methods:

```
inf.subtype(t1, t2) // <:
inf.and(c1, c2) // ^
inf.or(c1, c2) // ∨
inf.implies(c1, c2) // →
inf.not(c) // ¬
```

t_1, t_2 represent types and c, c_1, c_2 represent type constraints. Each method corresponds to each kind (comments on the right) of the type constraints in Listing 3. For example, constraint (3) in Listing 4 is described as

```
Constraint cst3 = inf.or(
    inf.and(inf.subtype(t_y, floatTy), inf.subtype(strTy, floatTy), inf.
        subtype(floatTy, t_plus)),
    inf.and(inf.subtype(t_y, strTy), inf.subtype(strTy, strTy), inf.subtype
        (strTy, t_plus)));
```

`Constraint` is the class representing type constraints. `t_y` and `t_plus` refer to type variables `t_y` and `t_plus` created by `inf.typevar`. `floatTy` and `strTy` are primitive types created by `inf.type`.

Finally, the described type constraint is given to `InferType` for type inference by

```
inf.add(cst3);
```

The other type constraints in Listing 4 can be described and given to `InferType` in a similar manner.

3.1.2 Encoding subtype-related typing rules for constraint solving

The compiler writer encodes the rest of the typing rules, which are not used in Section 3.1.1 and are called subtype-related typing rules, and give them to `InferType` for type inference. A subtype-related typing rule in `InferType` is an implication from zero or more premises to one conclusion encoded as

```
inf.ruleBuilderdeclare(conclusion).when(premise, premise, ...)
```

A premise or conclusion is one single subtype relation created by `inf.subtype`.

```
premise, conclusion := inf.subtype(t1, t2)
```

Method `declare` specifies the conclusion; the following method `when` specifies the premises. In `InferType`, all type variables involved in the premises must be included in the conclusion of an encoded subtype-related typing rule. For example, ST-Arrow in Figure 1 is encoded as

```
inf.ruleBuilder
  .declare(inf.subtype(inf.type("arrow", t11, t12), inf.type("arrow", t21
    , t22)))
  .when(inf.subtype(t21, t11), inf.subtype(t12, t22));
```

`t11, t12, ...` are type variables created by `inf.typevar`.

A subtype-related typing rule without premises can also be encoded as

```
inf.ruleBuilderdeclare(conclusion).always()
```

Method `always` specifies that the conclusion holds without a condition, which is a syntax sugar for method `when` without argument. For example, ST-Int-Float is encoded as

```
inf.ruleBuilderdeclare(inf.subtype(intTy, floatTy)).always();
```

Encoded subtype-related typing rules by `inf.ruleBuilder` are implicitly given to `InferType` for type inference.

3.1.3 Solving type constraints based on the encoded subtype-related typing rules

After describing type constraints and encoding subtype-related typing rules, the compiler writer calls method `solve` to solve the given type constraints based on the given encoded subtype-related typing rules.

```
Map<Typevar, Type> solution = inf.solve();
```

`Typevar` is a subclass of `Type` representing only type variables. By calling this method, `InferType` computes if there is a binding of the involved type variables in the given type constraints to concrete types so that all type constraints are evaluated to be true under the

23:8 InferType

given encoded subtype-related typing rules by replacing the type variables with their concrete types. If yes, InferType outputs that binding as the type inference results. For Listing 2, it will output a Map object representing the following binding.

```
{tf ↦ arrow<str, str>, tx ↦ str, tfx ↦ str,  
ty ↦ str, tplus ↦ str, te ↦ arrow<str, str>}
```

Otherwise, it indicates that the constraint solving fails such that InferType could not find a binding that makes all the given type constraints hold. InferType then can provide a set of type variables appearing in the ill-typed constraints.

```
if (!inf.untypableTypevars.isEmpty()) { // type error occurred  
    Set<Typevar> untvarts = inf.untypableTypevars();  
    // further locate type errors  
}
```

The compiler writer can manually track the type variables to program expressions such as recording them into a Map object during AST traversal:

```
locTrack.put(tv, String.format("at file %s: line %s, column %s",  
    fileName, lineNumber, columnNum));
```

where locTrack is the Map object that associates a type variable tv to its program location. Here, a value in the Map object is a string representation of a program location, where fileName, lineNumber and columnNum are managed by the compiler writer. Then she is expected to use the untypable type variables returned by InferType to retrieve the expressions that causes the type error, and further generate a type error message.

3.1.4 Declaring User-Defined Types

InferType also supports user-defined types such as `struct` in the C language and classes in object-oriented languages. During AST traversal, a compiler write can declare subtype-related typing rules for those user-defined types, and give those declarations to InferType. A compiler writer can describe new types, encode new subtype-related typing rules and give them to InferType at any phrase before calling `inf.solve` for constraint sovling.

For example, suppose that a compiler writer is implementing a type inference engine for a language supporting class definitions. When traversing an AST node for a class definition such as

```
class Car(Vehicle):  
    ...
```

which defines a class `Car` that inherits another class `Vehicle`, the compiler writer describes a new type for that class as

```
Type carTy = inf.type("car");
```

She then encodes a new subtype-related typing rule specifying the inheritance as

```
inf.ruleBuilder.declare(inf.subtype(carTy, vehicleTy)).always();
```

where `vehicleTy` is the type for class `Vehicle` described as `inf.type("vehicle")`. Like other subtype-related typing rules, the encoded subtype-related typing rule here is also implicitly given to InferType for type inference. InferType will perform constraint solving considering this encoded subtype-related typing rule when method `inf.solve` is called.

3.2 Translation to Z3

InferType performs constraint solving using the Z3 SMT solver. When method `inf.solve` is called, it translates the type constraints and subtype-related typing rules given by the compiler writer to Z3 formulae. The translated Z3 formulae are all asserted to check satisfaction by invoking the Z3 SMT solver to find if there is an interpretation of variables that makes all the asserted formulae true.

3.2.1 Translating types and type constraints

Types are translated to Z3 constants over a generated Z3 data type declaration. The data type declaration is generated by accumulating all the `Type` objects created by method `inf.type` included in the type constraints and encoded subtype-related typing rules given to InferType. This generated data type declaration represents the type definition of the target language. For Mini- λ , the data type declaration for `ztype` is generated as (represented using the SMT-LIBv2 [3] syntax)

```
(declare-datatypes () ((ztype
  (Int) (Float) (Str)
  (Arrow (ArrowP1 ztype) (ArrowP2 ztype)))))
```

It corresponds to t in Listing 1. `Int`, `Float` and `Str` are translated Z3 constructors representing the primitive types. `Arrow` represents the composite type `arrow`, where `ArrowP1` and `ArrowP2` are generated accessors for `Arrow`, which indicates that arrow types take two arguments. A type variable t_k is translated to a Z3 constant z_k ranging over `ztype`.

```
(declare-constant z_k ztype)
```

A composite type `arrow< t_k , int>` is translated to an application of `ztype` constructors.

```
(Arrow z_k Int)
```

The given type constraints are straightforwardly translated to Z3 boolean propositional formulae by the following translation function.

```
trans{inf.subtype(t1, t2)} = (zsubtype z1 z2)
trans{inf.and(c1, c2)} = (and trans{c1} trans{c2})
trans{inf.or(c1, c2)} = (or trans{c1} trans{c2})
trans{inf.implies(c1, c2)} = (=> trans{c1} trans{c2

```

c , c_1 and c_2 represent type constraints. The subtype relation $<$: is declared as `zsubtype` in Z3.

```
(declare-fun zsubtype (ztype ztype) Bool)
```

It specifies that `zsubtype` is a Z3 function that takes two arguments of the data type `ztype` and returns a boolean value. Logical connectives in type constraints are directly translated to Z3 logical operators. \Rightarrow is the logical implication operator in Z3. For instance, `cst3` in Section 3.1.1 is translated to

```
(or (and (zsubtype z_y Float) (zsubtype Str Float) (zsubtype Float z_plus
  ))
  (and (zsubtype z_y Str) (zsubtype Str Str) (zsubtype Str z_plus)))
```

z_y and z_{plus} in Z3 refer to `t_y` and `t_plus` in Java.

3.2.2 Translating subtype-related typing rules

The encoded subtype-related typing rules are processed by InferType to compute subtype relations, which are then translated into Z3 formulae. Since InferType implicitly assumes the reflexivity and transitivity rules, it also considers these rules when computing subtype relations. We borrow the idea of this process from Typette [14].

For each primitive type and composite type, InferType enumerates all its subtypes and super types in accordance with given subtype-related typing rules. It may also generate type constraints that those subtypes and super types must hold. We below mention how InferType enumerates subtypes. InferType does the same for enumerating super types.

Suppose that InferType enumerates subtypes for a target type $type_T$. Since InferType assumes the reflexivity rule, it first obtains this subtype relation: $type_T$ is a subtype of $type_T$. Next, suppose that the following subtype-related typing rule is given:

$$\frac{premise_1}{type_1 <: type_2}$$

Here, $type_k$ is either a primitive type, a composite type, or a type variable. If $type_2$ and $type_T$ can be *unified*, that is, they are lexically equivalent by replacing the type variables in $type_2$ and $type_T$ with other type variables, primitive types, or composite types, then InferType obtains the following subtype relation: $type_1$ is a subtype of $type_T$ when $premise_1$ holds, where the occurrences of some type variables in $type_1$ and $premise_1$ are replaced as they are for the unification between $type_2$ and $type_T$. When enumerating subtypes and super types for composite types, InferType only enumerates for the most generic forms of composite types, where all type arguments are type variables. For example, InferType computes subtypes and super types for $\text{arrow}\langle t_1, t_2 \rangle$, where t_1 and t_2 are type variables; but it does not compute subtypes or super types for an individual type such as $\text{arrow}\langle \text{int}, \text{int} \rangle$ or $\text{arrow}\langle \text{int}, t_1 \rangle$. Besides, InferType does not consider the reflexivity rule when enumerating subtypes and super types for composite types.

By this enumeration, in the case of Mini- λ , InferType enumerates float (by reflexivity) and int (by ST-Int-Float) as subtypes of float . By ST-Arrow, InferType enumerates $\text{arrow}\langle t_{11}, t_{12} \rangle$ as a subtype of $\text{arrow}\langle t_1, t_2 \rangle$ under premises $t_1 <: t_{11}$ and $t_{12} <: t_2$.

Furthermore, InferType considers the transitivity rule. Given the following rules, if $type_2$ and $type_3$ are unified,

$$\frac{premise_1}{type_1 <: type_2} \quad \frac{premise_2}{type_3 <: type_4}$$

InferType combines the two rules to derive the following new rule:

$$\frac{premise_1 \quad premise_2}{type_1 <: type_4}$$

where the occurrences of some type variables in $type_1$, $type_4$ and $premise_1$, $premise_2$ are replaced as they are for the unification between $type_2$ and $type_3$. They are replaced according to the substitution for a most general unifier [20], which is a complete and minimal substitution. Existential quantifiers are added¹ to the type variables that are included in $premise_1$ and $premise_2$ but not in $type_1$ or $type_4$. When $premise_1$ and $premise_2$ include $type_i <: type_j$ and $type_j <: type_k$, and all the type variables in $type_j$ are not included except $type_i$, $type_j$, and $type_k$, InferType combines $type_i <: type_j$ and $type_j <: type_k$ and transforms them into $type_i <: type_k$ by the transitivity rule.

¹ When an existential quantifier is included in the resulting Z3 formulae, Z3 might fail to correctly derive types. This is a limitation of InferType but it is out of the scope of this paper.

The derived new subtype-related typing rules are used to enumerate a subtype of the target type $type_T$. Furthermore, it may be combined with another rule to derive another new rule by the transitivity rule. InferType iterates this to enumerate all subtypes of $type_T$. It tries all possible combinations between subtype-related typing rules including the derived ones.

For example, suppose that InferType is computing a subtype of `intArray` (which is described as a primitive type `inf.type("intArray")`) and given two subtype-related typing rules:

$$\frac{t_1 <: t_2 \quad t <: int}{list<t_1> <: array<t_2>} \text{(a)} \quad \frac{t <: int}{array<t> <: intArray} \text{(b)}$$

InferType first unifies `array<t_2>` and `array<t>`, and finds a substitution $\{t_2 \mapsto t\}$ for that unification. Then it combines the rules and derives:

$$\frac{t_1 <: t \quad t <: int}{list<t_1> <: intArray} \text{(a and b)}$$

Furthermore, InferType implicitly applies the transitivity rule and obtains:

$$\frac{t_1 <: int}{list<t_1> <: intArray} \text{(a and b)}$$

This new rule is considered for the subtype enumeration, and InferType obtains `list<t_1>` as a subtype of `intArray` under the premise $t_1 <: int$.

The derived new rule may be combined with existing other rules, and another new rule may be derived from that combination by the transitivity rule. The iteration of this combining may not terminate within finite steps. For example, this iteration never terminates if the given subtype-related typing rules include a self-recursive subtype relation such as:

$$\frac{t_1 <: t_2 \quad t <: array<t>}{list<t_1> <: array<t_2>} \text{(a)} \quad \frac{t <: array<t>}{r} \text{(r)}$$

Here, the conclusion of `r` includes a self-recursive subtype relation such that a type is a subtype of an array type of itself. InferType combines `a` and `r`, and derives:

$$\frac{t_1 <: t_2 \quad t <: array<array<t_2>>}{list<t_1> <: array<array<t_2>>} \text{(a and r)}$$

InferType will further combine `a` and `r` and `r`. It will then infinitely combine and derive new rules. It is the InferType users' responsibility to ensure that the given subtype-related typing rules do not cause infinite iteration.

After enumerating all subtypes and super types for each primitive and composite type, InferType generates Z3 formulae representing type constraints for these subtype relations. Suppose that, for a target type $type_T$, its subtypes are $subtype_1$ when $premise_1$ holds, $subtype_2$ when $premise_2$ holds, ..., InferType then generates the following Z3 formula:

```
(forall ((z ztype) (z0 ztype) (z1 ztype) ...)
  (=> (zsubtype z typeT)
    (or (and (= z subtype1) premise1)
        (and (= z subtype2) premise2)
        ...)))
```

`(z ztype)` expresses that a Z3 variable `z` represents a type in the target language such as Mini- λ . `zsubtype` is the Z3 function returning true if the first argument is a subtype of the second argument. In the above formula, `z0`, `z1`, ... are Z3 variables representing type variables appearing in $type_T$, $subtype_1$, $subtype_2$, ... and $premise_1$, $premise_2$, `=>` is an implication operator in Z3. For example, for the subtypes of `arrow` in Mini- λ , InferType generates the following formula:

23:12 InferType

```
(forall ((z ztype) (z21 ztype) (z22 ztype))
  (=> (zsubtype z (Arrow z21 z22))
    (and (= z (Arrow (ArrowP1 z) (ArrowP2 z)))
      (zsubtype z21 (ArrowP1 z))
      (zsubtype (ArrowP2 z) z22)))))
```

This reads as, for all z , $z21$ and $z22$, if z is a subtype of $\text{arrow}(z21, z22)$, then z is an arrow type, $z21$ is a subtype of the first argument of z , and the second argument of z is a subtype of $z22$. InferType also generates a Z3 formula for the super types of `arrow`.

Finally, all translated Z3 formulae are asserted to find if there is an interpretation of variables that makes all the translated Z3 formulae true. If Z3 can find such an interpretation, InferType retrieves and translates that interpretation back to a binding of type variables to concrete types. This binding is returned to the compiler writer. Otherwise, InferType uses the unsat core extraction feature of Z3 to obtain a list of unsat translated type constraints. It then returns type variables in those unsat translated type constraints by calling `inf.untypableTypevars` in Section 3.1.3.

3.3 Optimizing Constraint Solving for Deeply Nested Types

Using Z3 in a straightforward way as in Section 3.2 works with respect to performance in common cases such as Listing 2. However, the constraint solving sometimes becomes extremely slow: we observe that the constraint solving time increases exponentially when the programs contain deeply nested types.

Consider another program in Mini- λ given in Listing 5.

```
fun f2. fun f. fun x. f2(f)(x)
  (fun g. g)
  (fun y. y + "a")
```

Listing 5 Another program with nested types in Mini- λ

This program extends Listing 2 by adding an outer function with parameter `f2`. The body of the function with parameter `x` is a function application, whose argument is `x` and the called function is another function application `f2(f)`. The function with parameter `f2` is then applied to the function with parameter `g`; its resulting value is applied to the function with parameter `y`. The type of parameter `f2` is supposed to be inferred as `arrow<arrow<str, str>, arrow<str, str>>`, which we call a nested type (arrow of arrow). In this paper, a nested type is a composite type that takes composite types as arguments. This program can be further modified by defining another outer function `f3` to create a larger nested type, where the type of `f3` is supposed to be inferred as a nested arrow of arrow of arrow type. We create the programs containing `f4` and `f5` by adding more outer functions for larger nested types. By our testing, a straightforward translation to Z3 in Section 3.2 performed constraint solving for the programs with `f3` in 0.18s, `f4` in 14.07s and `f5` in 179m15.35s!

We expect that this slow down arises from the increasing search space for SMT solving. For example, when a type variable is constrained to be a subtype (or super type) of a type taking arrow types as arguments in a given type constraint, that type variable ranges over increasingly many possible types by the size of arrow types in the arguments. In Mini- λ , such a type variable ranges over possible types `int`, `float`, `str`, `arrow<int, int>`, `arrow<arrow<int, ...>, int>`, ... by the increasing size of the arrow types in the arguments. When Z3 solves the given type constraints, it tries to search for a possible solution by instantiating the type variables over their possible types.

To mitigate the performance bottleneck of constraint solving containing deeply nested types in Z3, we develop a pre-process for InferType. It first computes structures for the type variables involved in the given type constraints. We call these structures *shapes*. Then it generates and asserts extra Z3 formulae based on the computed shapes. They reduce the search space of the involved type variables since type variables range over only limited depth of nested types (or primitive types) specified by the computed shapes.

A shape is either a shape variable, a symbol `*`, or a nested structure starting with a symbol `?`.

```
shape := shapevar | * | ?<shape, shape, ...>
```

`*` represents only primitive types; it can never be any composite type. `?` represents composite-type names such as “`arrow`” in Mini- λ , which constructs a nested structure for shapes by taking other shapes as arguments. For example, suppose that the shape of a type variable in Mini- λ is pre-computed as `?<*, *, ...>`, that type variable would range over only arrow types with primitive types as arguments, which are `arrow<int, int>, arrow<int, str>, ...`

3.3.1 Pre-Computing Shapes

InferType’s pre-process first computes shapes for the type variables involved in the given type constraints generated from syntax-related typing rules during AST traversal. Subtype-related typing rules are not considered when computing shapes.

Given the type constraints from syntax-related typing rules, our pre-process first converts the involved types to shapes by the following function:

```
shapeof{tk} = sk
shapeof{primType} = *
shapeof{compTypeName<t1, t2, ...>} = ?<shapeof{t1}, shapeof{t2}, ...>
```

A type variable is converted to a shape variable. A primitive type is converted to `*`. A composite type is converted to a nested structure with `?`. For example, a type `arrow<tk, int>` is converted to a shape `?<sk, *>`.

Then, *shape equations* are derived from the given type constraints. A shape equation is a logical assertion with conjunctions and disjunctions over a binary equality between two shapes.

```
shape-eq := shape == shape
| shape-eq ∧ shape-eq
| shape-eq ∨ shape-eq
```

A binary equality `shape1 == shape2` specifies that `shape1` and `shape2` are lexically identical. Shape equations do not contain logical implication or negation. Shape equations are derived by the following function:

```
derive{t1 <: t2} = shapeof{t1} == shapeof{t2}
derive{¬ t1 <: t2} = TRUE
derive{c1 ∧ c2} = derive{c1} ∧ derive{c2}
derive{c1 ∨ c2} = derive{c1} ∨ derive{c2}
```

The input is the Negation Normal Form (NNF) converted from the given type constraints in the form of Listing 3. The NNF is converted to handle type constraints in logical implication and negation. A subtype relation `t1 <: t2` is derived to a binary equality `s1 == s2` between the converted two shapes. Note that no shape equations are derived from subtype relations in logical negation (represented as returning a logical `TRUE` literal). For example, a shape equation `¬ sh == *` cannot be derived even if a type constraint `¬ th <: int` is given. `sh` can be arbitrary because `th` can be any type except a subtype of `int`.

23:14 InferType

For instance, the shape equations in Listing 6 are derived from the given type constraints in Listing 4.

```
(1') sf == ?<sx, sfx>
(2') ?<sf, ?<sx, sfx>> == ?<?<sy, splus>, se>
(3') (sy == * ∧ * == * ∧ * == splus)
    ∨ (sy == * ∧ * == * ∧ * == splus)
```

Listing 6 Shape equations for the type constraints

Next, the derived shape equations are solved to find a binding of shape variables to shapes so that all those shape equations are satisfied. A shape equation consists of conjunction, disjunction, and equality. Disjunctions may make it time-consuming to solve shape equations. An equality such as $?<s_1> == ?<*>$ may need to run an expensive unification algorithm when solving shape equations.

To make the shape computation fast, InferType adopts an approximate approach. Each shape equation is first transformed into a disjunctive normal form (DNF). Then, each clause in the DNF is separately solved; a set of substitutions for shape variables is found so that the left and right operands of every $==$ operator included in that clause will be lexically identical after the substitutions are applied to the operands. These substitutions are found by unification, which we implement by the disjoint-set (union-find) algorithm. For example, when a clause of DNF is $?<s_1> == ?<*> \wedge s_2 == s_1$, a set of substitutions $\{s_1 \mapsto *, s_2 \mapsto s_1\}$ is found. Note that one arbitrary set of substitutions is found when there exists multiple valid sets of substitutions.

After the unification, the resulting set of substitutions are compared with the sets of substitutions for the other clauses of the shape equation. If all the sets of substitutions are identical, the first clause of the shape equation, which is a conjunction of equalities, is used in the next step. Otherwise, the whole shape equation is excluded in the next step. This makes our shape computation approximate. When valid sets of substitutions are not found for some clauses, the shape equation is also excluded.

Finally, the remaining clauses, which are not excluded in the previous step, are combined by conjunction, and they are solved. The substitutions for shape variables are found by unification as in the previous step. A shape variable is bound to a shape obtained by applying those substitutions to that shape variable. If a shape is not obtained, the shape variable does not have a binding.

For example, our approximate shape computation computes the following binding for Listing 6:

```
{sf ↦ ?<*, *>, sx ↦ *, sfx ↦ *, 
sy ↦ *, splus ↦ *, se ↦ ?<*, *>}
```

Note that some shape variables do not have a binding. For the type variables corresponding to those shape variables, InferType does not generate extra Z3 formulae in the next step in Section 3.3.2. Those type variables are not optimized for constraint solving. For example, in a target language, suppose that a composite type `arrow` is a subtype of a primitive type `object`, and an operator `!=` takes operands of `object` type. Then, suppose that a variable v_k is initialized with a value of an arrow type and v_k appears as an operand for `!=`. This will generate a shape equation $s_k == ?<*, *> \wedge s_k == *$, where s_k corresponds to a type variable t_k for the variable v_k . The initialization of v_k generates $s_k == ?<*, *>$, but the `!=` operator generates $s_k == *$. s_k does not have a binding since there is no concrete shape for s_k to satisfy that shape equation. Note that not all shape variables s_k lose a binding when an arrow type is a subtype of a primitive type `object`. They lose a binding only when

the type variables t_k corresponding to s_k appears in an expression where a syntax-related typing rule specifies that t_k is a subtype of `object`. Recall that shape computation does not consider subtype-related typing rules.

3.3.2 Reducing Search Space

After solving shape equations, InferType generates extra Z3 formulae that explicitly represent what types the involved type variables range over regarding the computed shapes for reducing the search space. If a shape is `*`, its type variable ranges over only all primitive types. If a shape is `?` with n arguments, its type variable ranges over all composite types with n arguments.

Given a type variable t_k and the computed shape `?<shape1, shape2, ..., shapen>` for its corresponding shape variable s_k , where $shape_i$ represents some computed shapes, InferType generates extra Z3 formulae for t_k as

```
(or (= zk (c1n shape1 shape2 ... shapen))
    (= zk (c2n shape1 shape2 ... shapen))
    ...)
(or (= shape1 (c1m shape11 shape12 ... shape1m))
    (= shape1 (c2m shape11 shape12 ... shape1m))
    ...)
(or (= shape2 p1) (= shape2 p2) ...)
...
(or (= shapen ...) ...)
```

z_k is the translated Z3 constant for t_k . c_i^n represents all composite types that take n arguments in the target language. The possible types for z_k are explicitly listed by logical disjunction. If $shape_j$ is a nested structure with `?` that takes m arguments, InferType will then generate extra Z3 formulae for $shape_j$ similar to the extra Z3 formulae for z_k . An example is $shape_1$ in the above code. $shape_{1i}$ represents its arguments. If $shape_j$ is `*`, InferType will then generate extra Z3 formulae specifying that $shape_j$ ranges over only primitive types. An example is $shape_2$ in the above code. p_i represents all primitive types in the target language.

For example, by the computed shape `?*>` for t_f in Listing 4, the extra Z3 formulae are generated as

```
(or (= zf (Arrow sh_1, sh_2)))
(or (= sh_1 Int) (= sh_1 Float) (= sh_1 Str))
(or (= sh_2 Int) (= sh_2 Float) (= sh_2 Str))
```

They specify that z_f ranges over only arrow types with arguments of primitive types. `?` here corresponds to composite type names that take two arguments, which only “arrow” matches in Mini-λ. The arguments `sh_1` and `sh_2` are generated Z3 constants.

Suppose that Mini-λ defined list types `list<t>` and array types `array<t>`, and also that the shape of a type variable t_h were computed as `?<?<*>>`. Then InferType would generate the extra Z3 formulae as:

```
(or (= zh (List sh_4)) (= zh (Array sh_4)))
(or (= sh_4 (List sh_5)) (= sh_4 (Array sh_5)))
(or (= sh_5 Int) (= sh_5 Float) (= sh_5 Str))
```

z_h is the translated Z3 constant for t_h . sh_4 and sh_5 are generated Z3 constants for the arguments. `?` here corresponds to composite type names that take one argument, which “list” or “array” matches.

Note that asserting extra Z3 formulae generated from some approximately computed shapes in our shape computation might cause type inference failure, that is, the constraint solving would result in unsat even though the types in the given type constraints could be

23:16 InferType

inferred. When our shape computation approximately computes some shapes by excluding some shape equations, InferType would encounter this limitation if that type variable in the given type constraints does not have an inferred type with that approximately computed shape.

For example, consider the following type constraints in the Mini- λ extended with type `object` where an arrow type is a subtype of the `object` type:

```
(21) arrow<int, int> <: tc
(22) object <: tc ∨ td <: tc
(23) object <: td
```

The (only) valid binding that satisfies the type constraints is $\{t_c \mapsto \text{object}, t_d \mapsto \text{object}\}$. The shape equations are derived as:

```
(21') ?<*, *> == sc
(22') * == sc ∨ sd == sc
(23') * == sd
```

By applying the approximate shape computation, (22') will be excluded because it finds two different substitutions $\{s_c \mapsto *\}$ and $\{s_d \mapsto s_c\}$ by unification over the clauses of DNF. The approximate shape computation then only computes (21') and (23'), and outputs $\{s_c \mapsto ?<*, *>, s_d \mapsto *\}$. The constraint solving will then fail because t_c will be restricted to range over only arrow types by the extra Z3 formulae generated from the approximately computed shape $?<*, *>$ for s_c . On the other hand, a precise shape computation by considering (22') would discard the shapes for s_c and s_d , and then the type constraints can be correctly solved.

To handle this limitation for InferType producing correct type inference, in our implementation, when InferType cannot infer the types with the pre-process, it will recover the constraint solving once again without the pre-process. In the second run of the recovery, there is no extra Z3 formula from the computed shapes and all the constraint solving is not optimized. Note that if the given type constraints contain type errors, both the first and second run will result in unsat. InferType can produce correct type inference instead of producing some wrongly inferred types, though it will take extra time by running 2 times when our approximate shape computation encounters this limitation.

3.3.3 Discussion

InferType's shape computation is sound, that is: If InferType can compute a binding that satisfies the given type constraints with the asserted extra Z3 formulae from the shape computation, then that binding must be one of the bindings that satisfies the original type constraints. Since InferType asserts the extra Z3 formulae by conjunction with the given original type constraints, if InferType computes a binding for satisfaction, then that binding must also satisfy the original type constraints. Although InferType might not be able to compute some of the bindings that satisfy the original type constraints because the possible inferred types for the type variables are restricted by the computed shapes, a computed binding by InferType must be a valid solution. InferType is guaranteed to never produce a wrongly inferred type if a program is ill-typed.

InferType's shape computation is not complete in general, where the completeness of the shape computation is defined as: InferType can compute a binding that satisfies the type constraints with the asserted extra Z3 formulae if the original type constraints can be satisfied. InferType's shape computation would be complete under the following assumption: all subtype relations hold only between types with the same structure. In such a target language, if a type variable is constrained to be a subtype or super type of a type, then the

structure of the inferred type for that type variable must be lexically identical to the structure of that type. For instance, the shape computation for Mini- λ is complete. InferType's shape computation is not complete when the target language supports subtype relations between types with different structures. In practical cases, such languages often support a type called `object` which is a super type of any type. For example, suppose that the following type constraints are given in a target language where the only common super type of arrow and map types is a primitive type `object`:

```
arrow<int, int> <: ta
map<int, int> <: ta
```

The type variable t_a can be only inferred as `object` for satisfaction. The shape computation will compute the shape for t_a as $?<*, *>$ from the derived shape equations:

```
?<*, *> == sa
?<*, *> == sa
```

However, InferType then could not find a binding for t_a with the asserted extra Z3 formulae because t_a cannot be inferred as a type with that computed shape.

InferType incorporates a workaround to avoid its failure of computing shapes when the target language supports subtype relations between types with different structures. Some shape variables lose their bindings in our shape computation if part of the shape equations cannot be solved by unification, while the other shape variables can still be computed and the constraint solving for those corresponding type variables can still be optimized. For the previous example containing arrow and map types, a common practice for InferType correctly computing shapes and further computing types is by programmers' type annotations. If a programmer annotates the expression for t_a as type `object`, a type constraint $t_a <: \text{object}$ will be collected by AST traversal and given to InferType together with the other type constraints. As we discussed in the last paragraph in Section 3.3.1, s_a will lose a binding in the shape computation because the derived shape equations including s_a cannot be solved by unification. InferType would then correctly perform type inference although the constraint solving for t_a would not be optimized.

When InferType wrongly computes some shapes and cannot infer types because of the incompleteness of the shape computation, InferType would encounter a fallback to recover the constraint solving as we mentioned in Section 3.3.2. Although the constraint solving is entirely unoptimized in such cases, InferType would be able to correctly compute a binding as the type inference result.

4 Experiment

We implemented a type inference engine for Mini- λ using InferType. We include it in our artifact on Zenodo ². The source code contains 291 LOC in Java with (manually counted) 48 LOC using InferType. Owing to InferType's optimization, the implemented engine performed type inference within around 0.18s for each program with nested types `f3`, `f4` and `f5` as shown in Listing 5, which is way faster than a straightforward translation to Z3.

To further demonstrate the usage of InferType and the effectiveness of InferType's optimization, we conduct several experiments by implementing a type inference engine using InferType for a subset of Python. We choose Python as the target language because we

² <https://zenodo.org/records/10981733>

 **Table 1** Dataset overview. Each value is the average number per program/project in each set.

	set1-Typpete	set2-fs	set3-CodeNet	set4-large	set5-ill
LOC	37	17	29	490	18
def	3	5	0.4	45	0.3
typevar	61	29	175	745	35
constraint	45	12	79	613	32

suppose that it is a good testbed for evaluating the usage capability of InferType when implementing a type inference engine with complex typing rules for a dynamic language like Python. Another concern of choosing Python is being aware of an existing work Typpete [14]. Typpete is one of the state-of-the-art type inference engines for Python using the Z3 SMT solver. Comparing a type inference engine by InferType with generated Z3 code with a manually written one would be a valuable evaluation for the usage of InferType.

Subpet is a re-implementation of a subset of Typpete using InferType. Typpete performs type inference by a manual encoding of type constraints and subtype-related typing rules to Z3 formulae while Subpet does it by using InferType. Subpet adopts a similar type system to Typpete. The main differences between Subpet’s and Typpete’s type systems are: The type system of Subpet is flow-insensitive; Subpet does not support some container types such as sequence types `sequence<t>`. Subpet contains 2,892 LOC in Java with (manually counted) 226 LOC using InferType. It uses the Python `ast` module to parse the source programs. The other code mainly consists of AST traversal and typing environment, class table definitions. Typpete encodes the type inference of Python programs into a MaxSMT problem by a manual encoding of type constraints and typing rules to Z3 formulae. It outputs type annotations for an input Python program. Typpete contains 6,189 LOC implemented in Python. Subpet is not so powerful as Typpete because Subpet is limited by the amount of our implementation resource. Besides, InferType encodes the type inference for a target language as a SMT problem while Typpete encodes it for Python into a MaxSMT problem. Typpete could possibly infer some principle types by manually encoding soft and hard constraints in syntax-related typing rules, while principle typing is generally not considered by InferType. Nevertheless, Subpet could be able to infer types for the programs in our dataset including real-world programs.

For the experiments, we build a dataset. It is publicly available in our artifact. The dataset contains five sets of Python programs including no type annotations:

The first set (set1-Typpete) contains 44 Python benchmark programs obtained from Typpete’s artifact ³. A few innocuous code modifications are made to overcome the implementation limitations of Subpet such as using explicit arguments instead of implicit ones. These modifications do not impact the functionality of the code.

The second set (set2-fs) is a series of 7 artificial Python programs containing larger nested types. One of these programs, for example, is manually created as

```
def f0(x):
    return x + x
def f1(f0, x):
    return f0(x)

f1(f0, 42)
```

³ <https://zenodo.org/records/3996670>

`f1` is a function taking functions as parameter. Its type is assumed to be inferred as a nested type. This Python program can be considered as a similar correspondence to Listing 5 in Mini- λ . The program can be further extended with more function definitions `f2`, `f3`, ... for larger nested types similar to what we showed in Mini- λ . We created the cases up to `f7`.

The third set (set3-CodeNet) contains 30 programs obtained from *Project CodeNet* [36]. Project CodeNet is a open-source, large-scale dataset containing solution programs to coding problems from online judge websites. It is expected to be diverse and representative for real-world programs. We downloaded its Python benchmarks, Version 1 at Sep. 6, 2023. We used the `grep` command to search programs containing keywords `graph` and `dijkstra`. Then we obtained 12 problem sets that had more than 10 hits of the two keywords. We used such keyword searching because we expect that there is a high potential of programmers defining data of nested types to solve graph coding tasks using the Dijkstra's algorithm [16]. Then 5 programs were randomly picked up from each problem set (each problem set contains 100 ~ 300 solution programs), resulting in 60 programs. 30 of the 60 programs were filtered by the implementation limitation of Subpet (mostly because of unknown library usage with respect to types). The remaining 30 programs were finally included in the third set.

The fourth set (set4-large) contains 4 larger programs/projects. The programs in this set are aimed to estimate the scalability of the shape computation in InferType. Two of the larger programs with 358 and 445 LOC were collected from the MOPSA project ⁴. A few code modifications were made to overcome the implementation limitations of Subpet. The other two projects were collected from Typete's artifact. One project includes 5 Python files with 312 LOC. The other project includes 9 Python files with 846 LOC. Although these programs and projects are not exceedingly large, each of them is more than 10 times larger than the average of the programs in the other sets by LOC.

The fifth set (set5-ill) contains 4 ill-typed programs. These programs are used to demonstrate the behaviour of handling type errors in InferType. Two of the ill-typed programs are benchmark programs collected from MOPSA. The other two ill-typed programs are from set1-Typete. We manually modified the programs to make them become ill-typed. A complete list of all the code changes in the dataset is included in our artifact.

Table 1 gives an overview of our dataset. Each value represents the average number per program/project in each set. LOC shows the average number of line of code. def shows the average number of function and method definitions. typevar and constraint show the average number of generated type variables and type constraints derived by syntax-related typing rules during Subpet's AST traversal.

4.1 Comparing Subpet with Typete

Firstly, we show the performance of type inference by Subpet and Typete. Table 2 shows the time results by running the two engines over set1-Typete and set2-fs in our dataset. We selected and included some of the larger programs by LOC in set1-Typete in Table 2a. The experiments were conducted on a Ubuntu 19.10 machine with 2.8 GHz Intel(R) Core(TM) i7-6700T CPU and 32GB RAM, equipped with OpenJDK 14, Python 3.9.6, Z3 4.12.2 and Typete 0.1. The results from Subpet were the average elapsed time of the later 10 runs by looping Subpet 20 times within the same JVM execution for each input Python program. This is in the interest of the slow JVM startup time. For fair comparison, we also modified Typete to loop 20 times for each input Python program and then took the average of the later 10 runs.

⁴ <https://mopsa.lip6.fr>

23:20 InferType

Table 2 Time for Type Inference by Typpeete and Subpet. †timeout (after 1,000s)

(a) set1-Typpeete.				(b) set2-fs.			
Name(.py)	LOC	Typpeete	Subpet	Name(.py)	LOC	Typpeete	Subpet
bellman_ford	61	2.54s	0.23s	prog_f1	8	0.85s	0.09s
crafting_challenge	132	3.32s	0.27s	prog_f2	11	1.17s	0.11s
deceitful	36	1.41s	0.12s	prog_f3	15	1.51s	0.11s
disjoint_sets	45	2.20s	0.20s	prog_f4	17	6.52s	0.17s
lattice	81	2.50s	0.16s	prog_f5	20	101.4s	0.28s
rockpaperscissor	79	2.87s	0.13s	prog_f6	23	†	1.04s
vehicle	92	2.65s	0.14s	prog_f7	27	†	3.08s

Both Subpet and Typpeete could infer the types for normal programs without nested types or with smaller nested types (programs in set1-Typpeete and prog_f1 to prog_f3 in set2-fs) within a reasonable time period. Subpet was faster than Typpeete mainly because of the language advantage: Subpet is implemented in Java and Typpeete is implemented in Python. However, Typpeete exposed its performance downside for prog_f4 to prog_f7 in set2-fs with deeply nested types. Subpet could finish their type inference much faster. Typpeete ran prog_f5 within 101.4s; it ran into timeout (†) for prog_f6 and prog_f7 after 1,000s. The performance advantage is not simply because Subpet is implemented in Java rather than Python. It mainly comes from InferType’s optimization. We will show our findings in the later subsection.

We also demonstrate the behaviour of handling type errors by Subpet. It can detect the type errors for all the ill-typed programs in set5-ill. Subpet tracks the types and type variables to program locations (including the file name, line number and column number) in a Map object during constraint generation in the AST traversal methods. By invoking InferType for type inference, Subpet uses the untypable type variables returned by InferType to retrieve the program locations and report them to the programmer, though a better readable text message is not given due to our limited implementation resource. For example, one of the ill-typed programs in set5-ill is written as

```
0 # arg_mismatch.py
1 def f(x):
2     return x
3
4 f()
```

This program is ill-typed because function `f` is called without argument but it is defined as taking one argument. Subpet reports the type error message as

```
type inference failed.
type error location:
arg_mismatch.py: line 4, column 0
```

Typpeete would report a similar type error message to the above by Subpet. Note that InferType can detect and help report type errors whenever the optimization is disabled or enabled. InferType would not wrongly infer types with our shape computation if the program is ill-typed.

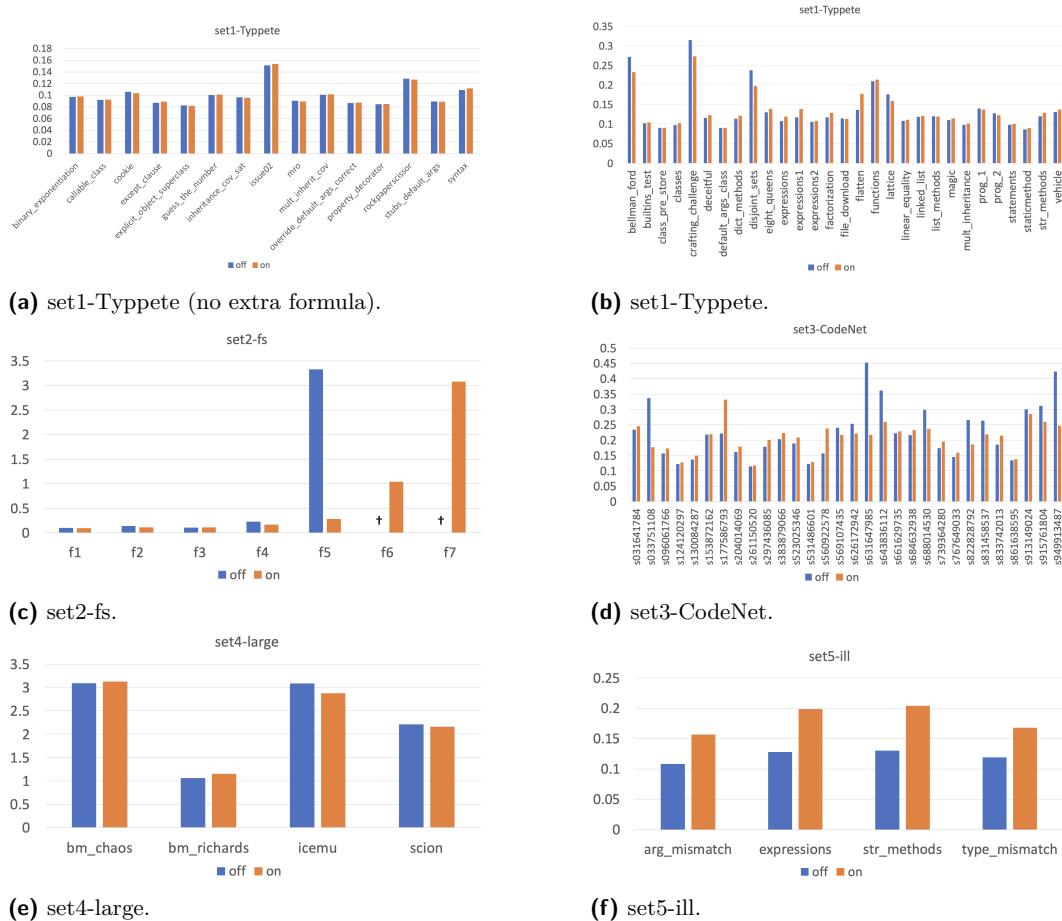


Figure 2 Time for Type Inference by disabling and enabling InferType’s shape computation. *off* columns are results by disabling shape computation. *on* columns are results by enabling shape computation. y-axes are the elapsed time in second. x-axes are program names. Figure 2a gives the results where the shape computation is enabled in *on* but extra Z3 formulae are not added in constraint solving. The other subfigures gives the results where extra Z3 formulae are added in *on*. †timeout (after 1,000s)

4.2 Validating the Effectiveness of InferType’s Optimization

We measure and compare the elapsed time of Subpet by disabling and enabling InferType’s shape computation in Section 3.3 to validate how our shape computation can affect the performance of type inference. Figure 2 shows the elapsed time for the programs in our dataset by disabling and enabling InferType’s shape computation. Each data column is the average elapsed time of the later 10 runs by looping Subpet 20 times. Column *off* and *on* show the elapsed time when InferType disables and enables the shape computation. When the shape computation is disabled, InferType performs type inference by a straightforward translation to Z3 shown in Section 3.2.

The results from Figure 2a show that InferType gives a small performance overhead by the shape computation. Figure 2a gives the results for part of the programs in set1-Typete. Particularly, extra Z3 formulae were not asserted in the constraint solving when the shape computation was enabled in the programs in Figure 2a for evaluating the overhead of the shape computation. Extra Z3 formulae were asserted for all the other programs when the shape computation was enabled in the other subfigures in Figure 2. In average, Subpet degraded the performance of type inference by 0.23% with the shape computation than

without the shape computation among all the programs in Figure 2a. Here and below, the performance rate is calculated as $(t_{off} - t_{on}) / t_{off}$ for each program, where t_{off} and t_{on} are the elapsed time when the shape computation is disabled and enabled. This 0.23% was the overhead of InferType’s shape computation for computing the shapes.

The results from Figure 2c show that InferType’s optimization can greatly mitigate the performance bottleneck for constraint solving containing deeply nested types. When the shape computation was enabled, Subpet performed type inference for all the programs in set2-fs within a reasonable time period. When the shape computation was disabled, Subpet performed much slower for `prog_f5`; it could not finish `prog_f6` or `prog_f7` within a time limit. These results were similar to those obtained by Typpe given in Table 2b. It indicates that the better performance by Subpet against Typpe was not simply because of the language advantage, but because of InferType’s optimization.

Our results demonstrate a tendency that InferType’s optimization has a higher potential to improve the performance of type inference for programs containing larger nested types. Subpet with the shape computation outperformed that without the shape computation in 8/29 (27.6%) programs among Figure 2b (set1-Typpe). Among these 8 programs that Subpet with the shape computation gave a better performance, it outperformed by an average of 8.87%. For example, program `bellman_ford` (containing a nested list of list type) and `crafting_challenge` (containing a nested dict of dict type) in Figure 2b had a better performance by 16.7% and 15.1%. Subpet with the shape computation outperformed that without shape computation in 11/30 (36.7%) programs among Figure 2d (set3-CodeNet). Among these 11 programs, it outperformed by an average of 40.9%. In the best case (`s631647985`), it accelerated the performance of type inference by 108.2%. The programs in set3-CodeNet have more nested types than programs in set1-Typpe because set3-CodeNet was collected on purpose by specific keyword search. Since set3-CodeNet has bias collected by specific keyword searching, we could not clearly show how frequently nested types are used in real-world Python programs. However, our results provide evidence that at least there are programs using nested types in wild Python programs. InferType is practically useful to help implement type inference engines such as Subpet to potentially handle such programs more efficiently. It is also observable that InferType’s optimization would not always give a better performance for the constraint solving with nested types. It sometimes degrades the performance even if the programs contain nested types. For example, the optimization degraded the performance of the two programs `s177586793` and `s560922578` by 49.6% and 51.6% in Figure 2d.

Our experiments empirically show the scalability of the shape computation to larger programs. Figure 2e presents the time results of type inference for the collected programs/-projects in set4-large. Subpet took around 2 seconds for type inference in average, which is 10 times longer than the average elapsed time for the programs in set1-Typpe and set3-CodeNet. InferType’s shape computation does not significantly boost or degrade the performance of type inference. These results show that it is practically feasible to apply our shape computation to larger real-world programs.

Figure 2f gives the elapsed time of type inference for the ill-typed programs in our dataset. Subpet took a longer time to find the type error in type inference when the shape computation was enabled. This is because InferType launches a second run of constraint solving if it cannot find a binding to satisfy the given type constraints in the first run for handling the limitation of our shape computation. Yet, Subpet could detect the type errors in all the ill-typed programs whenever the shape computation was disabled or enabled.

Table 3 Average number of computed shapes and average shape computation time. Each data cell shows the results by Approximate / Precise shape computation.

	set1-Typpe	set2-fs	set3-CodeNet	set4-large	set5-ill
shape	23 / 35	17 / 28	41 / 69	370 / 443	17 / 25
time	0.91ms / 74.3ms	0.71ms / 30.4ms	1.75ms / 434ms	68.7ms / 2.58s	0.69ms / 38.9ms

4.3 Assessing the Precision of the Approximate Shape Computation

We assess the precision of our approximate shape computation by counting the number of computed shapes for type variables, compared with an implemented, precise shape computation. Instead of computing only part of the shape equations in the approximate shape computation, the precise shape computation processes all the derived shape equations including disjunctions. It is implemented using Z3 by a straightforward translation from shape equations to Z3 formulae. For example, the shape equations in Listing 6 (which are derived from Listing 4) are translated to the following Z3 formulae to compute the shapes in the precise shape computation.

```
(= s_f (Q2 s_x s_fx))
(= (Q2 s_f (Q2 s_x s_fx)) (Q2 (Q2 s_y s_plus) s_e))
(or (and (= s_y Star) (= Star Star) (= Star s_plus))
         (and (= s_y Star) (= Star Star) (= Star s_plus)))
```

`s_?` represents translated Z3 constants for shape variables, which ranges over a Z3 datatype declaration `zshape` for shapes generated as

```
(declare-datatypes () ((zshape
  (Star)
  (Q2 (Q2P1 zshape) (Q2P2 zshape)))))
```

`Star` represents the symbol `*`. `Q2` represents the symbol `?` that takes two arguments, where `Q2P1` and `Q2P2` are generated accessors. The precise shape computation outputs a binding of shape variables to shapes by invoking Z3. InferType then generates and asserts extra Z3 formulae for type variables regarding the computed shapes to optimize constraint solving same as in Section 3.3.2. Our approximate shape computation is implemented in Java.

The approximate shape computation computes more than half of the shapes that the precise shape computation computes in average among all the programs in the dataset. The first row in Table 3 shows the average number of computed shapes of all programs in each dataset by the approximate and precise shape computation. The approximate shape computation computed 70.3%, 55.5%, 60.4%, 83.6% and 68.0% shapes of those by the precise one in each set of programs.

The approximate shape computation is much faster than the precise shape computation. The second row in Table 3 lists the average elapsed time of the two shape computations for all programs in each dataset. Same as the measuring before, the elapsed time of the shape computation for each program was the average of the later 10 runs by looping 20 times. The elapsed time of the approximate and precise shape computation differed in scale.

In our dataset, we did not find a program that had a faster elapsed time for type inference with the precise shape computation than approximate by Subpet, though the precise shape computation could compute more shapes in average and reduce the search space of more type variables for expected, faster constraint solving. This might suggest that our approximate shape computation is practically useful enough with respect to the performance, or the programs in our dataset are relatively small so that the precise shape computation could

not show its advantage. We could not show clear evidence to support these claims. Besides, we did not find a case such that Subpet performed a type inference failure caused by the limitation of the approximate shape computation as we discussed in Section 3.3.2. This is because Subpet’s syntax-related typing rules during AST traversal do not generate a type constraint like (22) given in Section 3.3.2, by excluding whose derived shape equation the approximate shape computation would imprecisely compute a shape. However, it is possible that our approximate shape computation encounters this limitation when developing other languages whose syntax-related typing rules generate such type constraints.

5 Related Work

Language development is enjoying significant growth in number and diversity both by academia and industry. Language workbenches such as MPS [45] and Xtext [11] are development environments that provide high-level mechanisms for implementing domain-specific languages [12]. Spoofax [18] allows language developers to write declarative specifications of language definitions to produce parsers, interpreters and editor plugins. Efftinge [10] presented a Java framework, Xbase, for implementing DSLs based on Xtext. Recent studies proposed a standardization of name resolution and type checking with a constraint-based approach integrated in Spoofax [1, 43]. Unlike existing language development toolkits such as Spoofax, our proposed tool produces constraint-based type inference.

Specifying and implementing type inference using constraints is an established approach. A number of existing type inference systems adopt constraint-based approaches because of the modularity of separating constraint generation and solving for providing high flexibility and extensibility [31, 35, 39, 38, 19, 27]. Our proposed tool adopts the constraint-based type inference for similar reasons to support various type systems of handling type inequalities such as subtyping. Besides the constraint-based approaches, one of the most traditional and influential type inference approaches is the Hindley-Milner type inference [15, 23, 6]. The Hindley-Milner type inference targets one particular type system and infers types by unification as the heart of its algorithm. Another approach is the bidirectional type checking [32, 26], which is regarded as local type inference, developed by carefully controlling the introduction and elimination of type variables for inferring parameter types. Turnstile [5] is a meta-language for creating typed languages supporting bidirectional type checking. Compared with Turnstile, InferType is designed for languages supporting global type inference by the constraint-based approach. Jones introduced wobbly types [30] to distribute over type constructors for handling GADT type inference using type annotations. Later, Pottier [34] proposed a stratified type inference with a pre-process of inferring shapes for propagating type annotations by local type inference. The idea of a two-strata type inference and the computation for shapes in this research are similar to those in [34, 46]. However, our shapes are automatically computed from the given type constraints without extra information like type annotations, and the computed shapes are used for optimizing constraint solving.

Satisfiability Modulo Theories (SMT) is an area of automated deduction for checking the satisfiability of first-order formulae with respect to logical theories [4, 3]. State-of-the-art SMT solvers include Yices [9], CVC5 [2], and Z3 [7]. InferType currently relies on Z3 for performing constraint solving. Translating InferType expressions to a higher-level language such as JavaSMT [17] can be a possible extension for enabling different SMT solvers. There have been works on improving the performance of SMT solvers in general such as pruning the search space by detecting symmetries in the input formulae [8]. Later, Niemetz [25] presented an approach for accelerating quantified constraint solving. The developed optimization in our proposal is rather a domain-specific approach to reducing the search space of type variables in SMT solving based on InferType’s shape computation.

SMT solving has been a critical part of several static analyses including automatic type inference. Swamy [42] presented the language F* with a dependent type and effect system using a combination of SMT solving and manual proofs. Vazou and Jhala [44] introduced LiquidHaskell, which is a refinement type-based verifier for Haskell using SMT solvers. InferType does not directly support such type systems. However, a compiler writer can handle complex types such as generics by manually resolving them (instantiating fresh type variables) before encoding them into InferType, though InferType would compute a concrete type instead of a generic type as the inferred type. Pavlinovic [28] presented an encoding of the OCaml type system to a weighted MaxSMT problem for localizing type errors when the type inference fails. InferType considers type errors by providing the type variables in a minimal set of unsat type constraints for helping locate ill-typed program expressions, while the generation of a text message is left to the compiler writer. Generating and customizing type error messages by InferType is treated as our future work. Hassan [14] proposed a type inference engine, Typete, that generates Python 3 type annotations by encoding type constraints as a MaxSMT problem using Z3. In this paper, we implemented a subset of Typete by using the proposed tool for the experiments. There are also emerging studies of statically typing Python programs based on other techniques [21, 13, 24, 29].

6 Conclusion

We presented InferType, a Java library for implementing constraint-based type inference. InferType performs constraint solving by translation to the Z3 SMT solver. Because the constraint solving in SMT may be exponentially slow by the increasing search space for large nested types, we developed an optimization technique for InferType to relieve the performance bottleneck for better practical usage. InferType pre-computes a structure of a type variable and reduces the search space of that type variable based on the pre-computed structure.

We demonstrated the usage of InferType and experimented the effectiveness of its optimization by implementing a type inference engine for a Python subset using InferType. We found that, the implemented engine had compatible performance of type inference compared with a state-of-the-art type inference engine for Python using Z3 with a manual encoding of Z3 formulae. InferType's optimization could greatly improve the performance for programs with deeply nested types. We also observed that InferType could potentially improve the performance of type inference for programs containing nested types. We believe that InferType is practically useful to help implement constraint-based type inference for language development.

References

- 1 Hendrik Antwerpen, Pierre Neron, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 49–60, January 2016. doi:10.1145/2847538.2847543.
- 2 Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A Versatile and Industrial-Strength SMT Solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442, Cham, 2022. Springer International Publishing.

- 3 Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, volume 13, page 14, 2010.
- 4 Clark Barrett and Cesare Tinelli. *Satisfiability Modulo Theories*, pages 305–343. Springer International Publishing, Cham, 2018. doi:10.1007/978-3-319-10575-8_11.
- 5 Stephen Chang, Alex Knauth, and Ben Greenman. Type Systems as Macros. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, pages 694–705, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3009837.3009886.
- 6 Luis Damas and Robin Milner. Principal Type-Schemes for Functional Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. Association for Computing Machinery. doi:10.1145/582153.582176.
- 7 Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.
- 8 David Déharbe, Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. Exploiting Symmetry in SMT Problems. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction – CADE-23*, pages 222–236, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- 9 Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 737–744, Cham, 2014. Springer International Publishing.
- 10 Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. Xbase: Implementing Domain-Specific Languages for Java. *SIGPLAN Not.*, 48(3):112–121, September 2012. doi:10.1145/2480361.2371419.
- 11 Moritz Eysholdt and Heiko Behrens. Xtext: Implement Your Language Faster than the Quick and Dirty Way. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, pages 307–309, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1869542.1869625.
- 12 Martin Fowler and R Parsons. Addison-Wesley signature, Domain specific languages . Massachussets, 2010.
- 13 google. pytype, 2021. URL: <https://google.github.io/pytype/>.
- 14 Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. MaxSMT-Based Type Inference for Python 3. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 12–19, Cham, 2018. Springer International Publishing.
- 15 R. Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969. URL: <http://www.jstor.org/stable/1995158>.
- 16 Donald B. Johnson. A Note on Dijkstra's Shortest Path Algorithm. *J. ACM*, 20(3):385–388, July 1973. doi:10.1145/321765.321768.
- 17 Egor George Karpenkov, Karlheinz Friedberger, and Dirk Beyer. JavaSMT: A Unified Interface for SMT Solvers in Java. In Sandrine Blazy and Marsha Chechik, editors, *Verified Software. Theories, Tools, and Experiments*, pages 139–148, Cham, 2016. Springer International Publishing.
- 18 Lennart C.L. Kats and Eelco Visser. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 444–463, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1869459.1869497.

- 19 Milod Kazerounian, Brianna M. Ren, and Jeffrey S. Foster. Sound, Heuristic Type Annotation Inference for Ruby. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2020, pages 112–125, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3426422.3426985.
- 20 Kevin Knight. Unification: A Multidisciplinary Survey. *ACM Comput. Surv.*, 21(1):93–124, March 1989. doi:10.1145/62029.62030.
- 21 Jukka Lehtosalo, Guido van Rossum, and Ivan Levkivskyi. mypy, June 2012. URL: <http://mypy-lang.org/>.
- 22 John R. Levine, Tony Mason, and Doug Brown. *Lex & Yacc (2nd Ed.)*. O'Reilly & Associates, Inc., USA, 1992.
- 23 Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978. doi:10.1016/0022-0000(78)90014-4.
- 24 Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. Static Type Analysis by Abstract Interpretation of Python Programs. In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, volume 166 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:29, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2020.17.
- 25 Aina Niemetz, Mathias Preiner, Andrew Reynolds, Clark Barrett, and Cesare Tinelli. Syntax-Guided Quantifier Instantiation. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 145–163, Cham, 2021. Springer International Publishing.
- 26 Martin Odersky, Christoph Zenger, and Matthias Zenger. Colored Local Type Inference. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 41–53, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/360204.360207.
- 27 Lionel Parreaux. The Simple Essence of Algebraic Subtyping: Principal Type Inference with Subtyping Made Easy (Functional Pearl). *Proc. ACM Program. Lang.*, 4(ICFP), August 2020. doi:10.1145/3409006.
- 28 Zvonimir Pavlinovic, Tim King, and Thomas Wies. Finding Minimum Type Error Sources. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 525–542, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2660193.2660230.
- 29 Yun Peng, Cuiyun Gao, Zongjie Li, Bowei Gao, David Lo, Qirun Zhang, and Michael Lyu. Static Inference Meets Deep Learning: A Hybrid Type Inference Approach for Python. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, pages 2019–2030, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3510003.3510038.
- 30 Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. *SIGPLAN Not.*, 41(9):50–61, September 2006. doi:10.1145/1160074.1159811.
- 31 Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.
- 32 Benjamin C. Pierce and David N. Turner. Local Type Inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, January 2000. doi:10.1145/345099.345100.
- 33 François Pottier. A Framework for Type Inference with Subtyping. *SIGPLAN Not.*, 34(1):228–238, September 1998. doi:10.1145/291251.289448.
- 34 François Pottier and Yann Régis-Gianas. Stratified type inference for generalized algebraic data types. *SIGPLAN Not.*, 41(1):232–244, January 2006. doi:10.1145/1111320.1111058.
- 35 Francois Pottier and Didier Remy. *The Essence of ML Type Inference*, pages 389–489. MIT press, January 2005. doi:10.7551/mitpress/1104.003.0016.
- 36 Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks, 2021. arXiv:2105.12655.

- 37 J Alan Robinson. Computational logic: The unification computation. *Machine intelligence*, 6:63–72, 1971.
- 38 Michael I. Schwartzbach. Type inference with inequalities. In S. Abramsky and T. S. E. Maibaum, editors, *TAPSOFT '91*, pages 441–455, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.
- 39 Tatsuro Sekiguchi and Akinori Yonezawa. A complete type inference system for subtyped recursive types. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, pages 667–686, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.
- 40 Jeremy Siek and Walid Taha. Gradual Typing for Objects. In Erik Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, pages 2–27, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
- 41 Richard M. Stallman. Bison: The Yacc-Compatible Parser Generator, 2015. URL: <https://api.semanticscholar.org/CorpusID:60543798>.
- 42 Nikhil Swamy, Cătălin Hrițcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent Types and Multi-Monadic Effects in F*. *SIGPLAN Not.*, 51(1):256–270, January 2016. doi:10.1145/2914770.2837655.
- 43 Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. Scopes as Types. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi:10.1145/3276484.
- 44 Niki Vazou, Eric L. Seidel, and Ranjit Jhala. LiquidHaskell: Experience with Refinement Types in the Real World. *SIGPLAN Not.*, 49(12):39–51, September 2014. doi:10.1145/2775050.2633366.
- 45 Markus Voelter and Vaclav Pech. Language Modularity with the MPS Language Workbench. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1449–1450. IEEE Press, 2012.
- 46 Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. OUTSIDEIN(X): modular type inference with local assumptions. *J. Funct. Program.*, 21:333–412, September 2011. doi:10.1017/S0956796811000098.

Qafny: A Quantum-Program Verifier

Liyi Li  

Iowa State University, Ames, IA, USA

Rance Cleaveland 

University of Maryland, College Park, MD, USA

Mingwei Zhu 

University of Maryland, College Park, MD, USA

Alexander Nicolellis 

Iowa State University, Ames, IA, USA

Yi Lee 

University of Maryland, College Park, MD, USA

Le Chang 

University of Maryland, College Park, MD, USA

Xiaodi Wu  

University of Maryland, College Park, MD, USA

Abstract

Because of the probabilistic/nondeterministic behavior of quantum programs, it is highly advisable to verify them formally to ensure that they correctly implement their specifications. Formal verification, however, also traditionally requires significant effort. To address this challenge, we present QAFNY, an automated proof system based on the program verifier Dafny and designed for verifying quantum programs. At its core, QAFNY uses a type-guided quantum proof system that translates quantum operations to classical array operations modeled within a classical separation logic framework. We prove the soundness and completeness of our proof system and implement a prototype compiler that transforms QAFNY programs and specifications into Dafny for automated verification purposes. We then illustrate the utility of QAFNY’s automated capabilities in efficiently verifying important quantum algorithms, including quantum-walk algorithms, Grover’s algorithm, and Shor’s algorithm.

2012 ACM Subject Classification Theory of computation → Program verification; Theory of computation → Quantum information theory

Keywords and phrases Quantum Computing, Automated Verification, Separation Logic

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.24

Related Version *Extended Version:* <https://arxiv.org/abs/2211.06411> [24]

Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact):*

<https://doi.org/10.4230/DARTS.10.2.12>

Software (Source Code): <https://zenodo.org/records/10529900>

Acknowledgements We thank Finn Voichick for his helpful comments and contributions during the work’s development. This paper is dedicated to the memory of our dear co-author Rance Cleaveland.

1 Introduction

Quantum computers can be used to program substantially faster algorithms compared to those written for classical computers. For example, Shor’s algorithm [48] can factor a number in polynomial time, which is not known to be polynomial-time-computable in the classical setting. Developing more and more comprehensive quantum programs and algorithms is essential for the continued practical development of quantum computing [11, 49]. Unfortunately, because quantum systems are inherently probabilistic and must obey quantum physics laws, traditional validation techniques based on run-time testing are virtually impossible to develop for large quantum algorithms. This leaves *formal methods* as a viable alternative for program checking, and yet these typically require a great effort; for example, four experienced researchers needed two years to formally verify Shor’s algorithm [38]. To alleviate the effort required for formal verification, many frameworks have been proposed to verify quantum algorithms [26, 56, 3, 59, 20, 16] using interactive theorem



© Liyi Li, Mingwei Zhu, Rance Cleaveland, Alexander Nicolellis, Yi Lee, Le Chang, and Xiaodi Wu;

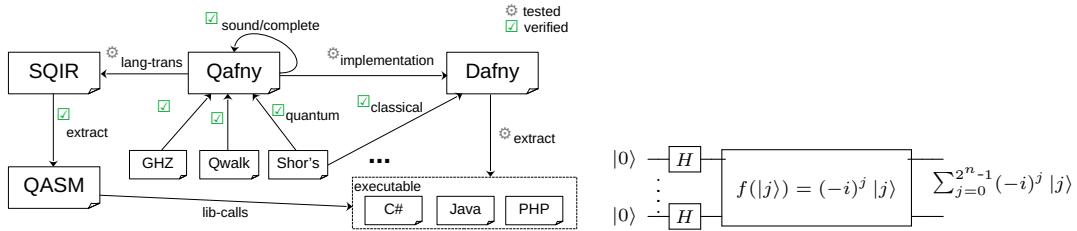
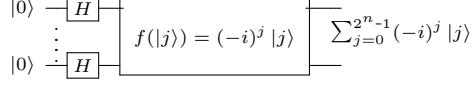
licensed under Creative Commons License CC-BY 4.0

38th European Conference on Object-Oriented Programming (ECOOP 2024).

Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 24; pp. 24:1–24:31



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

**Figure 1** Dafny Development Stages/Key Aspects.**Figure 2** State Preparation Circuit.

provers, such as Isabelle, Coq, and Why3, by building quantum semantic interpretations and libraries. Some attempts towards proof automation have been made by creating new proof systems for quantum data structures such as Hilbert spaces; however, building and verifying quantum algorithms in these frameworks are still time-consuming and require great human effort. Meanwhile, automated verification is an active research field in classical computation with many proposed frameworks [17, 42, 27, 39, 37, 18, 50, 31, 44, 45, 21, 6] showing strong results in reducing programmer effort when verifying classical programs. None of the existing quantum verification frameworks utilize these classical verification infrastructures, however.

We present QAFNY, a framework that enables programmers to develop and verify *quantum programs* based on quantum program semantics and classical automated verification infrastructure. It has several elements (Figure 1). The core is a strongly typed, flow-sensitive imperative quantum language QAFNY, admitting a classical separation-logic-style proof system, in which users specify quantum programs and input the properties to be verified as pre- and post-conditions and loop invariants, such as GHZ, Quantum Walk, and Shor’s algorithm. QAFNY programs and specifications are verified via translation to a classical Hoare/separation logic framework implemented in the Dafny program verifier [22]. QAFNY programs may also be compiled into quantum circuits and run on a quantum computer via the QAFNY to SQIR and SQIR to OpenQASM 2.0 [7] compilers in our technical report (TR) [24] C.6. Quantum programs can be components of *hybrid classical-quantum* (HCQ) programs, so the compiled QAFNY code can also be a library function called by an HCQ program defined. For example, one can extract the compiled Dafny program to a programming language, such as C#, PHP, and Java, and utilize quantum programs compiled from QAFNY to OpenQASM [21].

A key component of the design of QAFNY is to encode quantum states as array-like structures and quantum operations as aggregate array operations on the states. In Figure 2, a quantum state in superposition $\psi = \sum_{j=0}^{2^n-1} 1 |j\rangle$ is prepared by applying a Hadamard gate to each qubit. QAFNY treats ψ as an array containing 2^n elements, one for each indexed basis element in ψ . Each element is a pair of a complex and a natural number (computational basis, essentially a bitstring). For example, Bell pair $\frac{1}{\sqrt{2}} |00\rangle + \frac{1}{\sqrt{2}} |11\rangle$ can be thought of as a two-element array, with two pairs: $(\frac{1}{\sqrt{2}}, |00\rangle)$ and $(\frac{1}{\sqrt{2}}, |11\rangle)$, where the first one is a complex number and the second one is a bitstring that can be represented as a natural number. Applying a quantum oracle ($f(|j\rangle) = (-i)^j |j\rangle$) on ψ , which evolves each indexed element $1 |j\rangle$ to $(-i)^j |j\rangle$, is similar to an array map function that applies $f'(\alpha_j, j) = ((-1)^j \alpha_j, j)$ to each element j in the 2^n -array. The design and analysis of many quantum algorithms leverage the representation of different groups of qubits in terms of classical arrays [13, 35, 47]. Besides the opportunities for automated reasoning provided by representing quantum states as arrays, QAFNY also uses language abstractions such as quantum conditionals and loops, which generalize quantum controlled gates to enable local reasoning in the presence of quantum entanglement. In the prior works above, the usual approach in reasoning about

quantum controlled gates such as `CNOT` and controlled- U gates is to transform these operations into a monolithic representation, such as a unitary matrix, not scaling up well for automated verification, because the relations among different entries, representing inductive relation among program constructs, are largely omitted. In QAFNY, reasoning about comprehensive constructs, such as controlled gates, amount to building a structural inductive relation among different parts, such as a quantum conditional and its subparts, through deliberately designed proof rules in Sections 3.4 and 4.4; such design permits automated local reasoning.

These designs pose several challenges: 1) quantum operations can be performed on any qubit positions, e.g., f above could apply on arbitrary bits in every bitstring j ; 2) performing automated local reasoning requires one to know which states and qubits can be excluded locally, but qubits can form entangled groups that are typically viewed as not separable; and 3) the QAFNY proof system should obey quantum physical laws, such as no-cloning and no quantum observer effects. To address these problems, we first introduce different types of quantum-state representations and special data structures (*loci* in Section 3) to partition qubits into disjoint entanglement groups for local reasoning. We then combine a flow-sensitive type system (Section 4.3) with our proof system, capable of 1) statically identifying the quantum state types and tracking entanglement group transformations and 2) performing type-guided quantum-state rewrites in the assertions and automated local operation reasoning on canonicalized quantum states, without violating quantum laws.

The paper's contributions are listed as follows.

- We present the QAFNY language, including a big-step semantics and flow-sensitive type system, which provides a simple way of enforcing quantum program properties, such as no-cloning and no observer breakdown. We also prove type soundness for QAFNY in Coq.
- The QAFNY type-guided proof system permits classical-array-operation views of quantum operations and captures the inductive behaviors of quantum conditionals and loops. Soundness and relative completeness are also proved in Coq. To the best of our knowledge, QAFNY provides the first proof-rule definitions for quantum conditionals and for-loops.
- We exhibit a prototype QAFNY to Dafny compiler as evidence of connecting quantum-program verification to classical Hoare/separation logic. We verify a number of quantum algorithms (Figure 16) with a high degree of automation. Sections 5.2 and 7 compares proof automation in QAFNY with other frameworks.
- We faithfully implement several algorithms, such as GHZ, Shor's, and quantum walk, as case studies in this paper to demonstrate how QAFNY can help programmers to efficiently verify quantum-algorithm implementations. The program operations of these examples are a high-level abstraction of the algorithms' quantum circuit-based description, while the QAFNY program state specifications are directly based on the algorithms' state representations based on Dirac notations.

2 Background

Here, we provide background information on quantum computing.

Quantum Value States. A quantum value state¹ consists of one or more quantum bits (*qubits*), which can be expressed as a two-dimensional vector $(\begin{smallmatrix} \alpha \\ \beta \end{smallmatrix})$ where the *amplitudes* α and β are complex numbers and $|\alpha|^2 + |\beta|^2 = 1$. We frequently write the qubit vector

¹ Most literature mentioned Quantum value states as *quantum states*. Here, we refer to them as quantum value states or quantum values to avoid confusion between program and quantum states.

as $\alpha|0\rangle + \beta|1\rangle$ (the Dirac notation [8]), where $|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ are *computational basis-kets*. When both α and β are non-zero, we can think of the qubit being “both 0 and 1 at once,” a.k.a. in a *superposition* [35], e.g., $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ represents a superposition of $|0\rangle$ and $|1\rangle$. Larger quantum values can be formed by composing smaller ones with the *tensor product* (\otimes) from linear algebra, e.g., the two-qubit value $|0\rangle \otimes |1\rangle$ (also written as $|01\rangle$) corresponds to vector $[0 \ 1 \ 0 \ 0]^T$. However, many multi-qubit values cannot be *separated* and expressed as the tensor product of smaller values; such inseparable value states are called *entangled*, e.g. $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, known as a *Bell pair*, which can be rewritten to $\sum_{d=0}^1 \frac{1}{\sqrt{2}}|dd\rangle$, where dd is a bit string consisting of two bits, each of which must be the same value (i.e., $d = 0$ or $d = 1$). Each term $\frac{1}{\sqrt{2}}|dd\rangle$ is named a *basis-ket* [35], consisting an amplitude ($\frac{1}{\sqrt{2}}$) and a basis vector $|dd\rangle$.

Quantum Computation and Measurement. Computation on a quantum value consists of a series of *quantum operations*, each acting on a subset of qubits in the quantum value. In the standard form, quantum computations are expressed as *circuits*, as in Figure 3a, which depicts a circuit that prepares the Greenberger-Horne-Zeilinger (GHZ) state [12] – an n -qubit entangled value of the form: $|\text{GHZ}^n\rangle = \frac{1}{\sqrt{2}}(|0\rangle^{\otimes n} + |1\rangle^{\otimes n})$, where $|d\rangle^{\otimes n} = \otimes_{d=0}^{n-1} |d\rangle$. In these circuits, each horizontal wire represents a qubit, and boxes on these wires indicate quantum operations, or *gates*. The circuit in Figure 3a uses n qubits and applies n gates: a *Hadamard* (H) gate and $n - 1$ *controlled-not* (CNOT) gates. Applying a gate to a quantum value *evolves* it. Its traditional semantics is expressed by multiplying the value’s vector form by the gate’s corresponding matrix representation: n -qubit gates are 2^n -by- 2^n matrices. Except for measurement gates, a gate’s matrix must be *unitary* and thus preserve appropriate invariants of quantum values’ amplitudes. A *measurement* operation extracts classical information from a quantum value. It collapses the value to a basis state with a probability related to the value’s amplitudes (*measurement probability*), e.g., measuring $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ collapses the value to $|0\rangle$ with probability $\frac{1}{2}$, and likewise for $|1\rangle$, returning classical value 0 or 1, respectively. A more general form of quantum measurement is *partial measurement*, which measures a subset of qubits in a qubit array; such operations often have simultaneity effects due to entanglement, i.e., in a Bell pair $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, measuring one qubit guarantees the same outcome for the other – if the first bit is measured as 0, the second bit is too.

Quantum Conditionals. Controlled quantum gates, such as controlled-not gates (CNOT), can be thought of as quantum versions of classical operations, where we view a quantum value as an array of basis-kets and apply an array map operation of the classical operation to every basis-ket. This is evident when using Dirac notation. For example, in preparing a two-qubit GHZ state (Figure 3a, also a Bell pair) for qubit array x , the H gate evolves the value to $\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|10\rangle$ (same as $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |0\rangle$). The quantum conditional maps the classical conditional `if` $(x[0]) \{x[1] \leftarrow x[1] + 1\}$ onto the two basis-kets, where the operation $x[1] + 1$ acts as a modulo 2 addition to flip $x[1]$ ’s bit. Here, we do not flip the $x[1]$ position in the first basis-ket ($\frac{1}{\sqrt{2}}|00\rangle$) due to $x[0] = 0$, and we flip $x[1]$ in the second basis-ket because of $x[0] = 1$. Such behaviors can be generalized to other controlled gates, such as the controlled- U gate appearing in Shor’s algorithm (Figure 6), where U refers to a modulo-multiplication operation. The controlled nodes (Boolean guards) in these quantum conditionals can also be generalized to other types of Boolean expressions, e.g., it can be a quantum inequality $((\kappa < n) @ x[i])$ that compares every basis vector of qubit array κ ’s value state with the number n and stores the result in qubit $x[i]$, and the controlled node queries $x[i]$ to determine if the conditional body is executed, more in Sections 4 and 6.2.

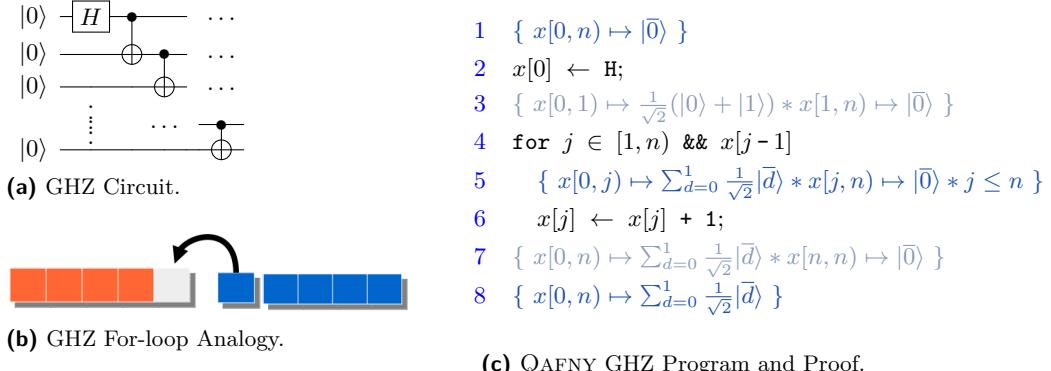


Figure 3 GHZ Examples. $x[t_1, t_2] \mapsto |\bar{0}\rangle$ means $\bar{0}$ is a bitstring of length $t_2 - t_1$. $x[t_1, t_2] \mapsto \sum_{d=0}^1 |\bar{d}\rangle$ means \bar{d} is a bitstring of length $t_2 - t_1$ and $d \in [0, 1]$ is a bit. $*$ is the separation conjunction. In (c), black parts are QAFNY programs, while blue and gray parts are QAFNY state predicates.

Quantum Oracles. Quantum algorithms manipulate input information encoded in “oracles,” which are callable black-box circuits. Quantum oracles are usually quantum-reversible implementations of classical operations, especially arithmetic operations. Their behavior is defined in terms of transitions between single basis-kets. We can infer its global state behavior based on the single basis-ket behavior through the quantum summation formula below. This resembles an array map operation in Figure 2. OQASM [23] is a language that permits the definitions of quantum oracles with efficient verification and testing facilities using the summation formula.

$$\frac{\forall j. |x_j\rangle \longrightarrow f(|x_j\rangle)}{\Sigma_j \alpha_j |x_j\rangle \longrightarrow \Sigma_j \alpha_j f(|x_j\rangle)}$$

No Cloning and Observer Effect. The *no-cloning theorem* suggests no general way of copying a quantum value. In quantum circuits, this is related to ensuring the reversible property of unitary gate applications. For example, the Boolean guard and body of a quantum conditional cannot refer to the same qubits, e.g., `if (x[1]) {x[1] ← x[1] + 1}` violates the property as $x[1]$ is mentioned in the guard and body. The quantum *observer effect* refers to leaking information from a quantum value state. If a quantum conditional body contains a measurement or classical variable updates, the quantum system breaks down due to the observer effect. QAFNY enforces no cloning and no observer breakdown through the syntax and flow-sensitive type system.

3 Qafny Design Principles: Locus, Type, and State

Here, we show the QAFNY fundamental design principles for quantum program verification. We use the GHZ example in Figure 3a to highlight these principles, with a proof outline in Figure 3c; x is initialized to an n -qubit `Nor` typed value $|0̄⟩$ (n number of $|0\rangle$ qubits). After preparing a superposition ($x[0] \leftarrow H$) for a single qubit $x[0]$ in line 2, we execute a *quantum for-loop* that entangles each pair of adjacent qubits in x to prepare the GHZ state. We can unroll each iteration of the loop as a quantum conditional (`if (x[j-1]) {x[j] ← x[j] + 1}`). When verifying the program in QAFNY, it is only needed to provide the program and specifications in `blue`, with the `grayed out` parts automatically inferred. We show critical features in our type-guided proof system, making the above verification largely automatic.

3.1 Loci, Types, and States

Figure 4 shows QAFNY *loci*, types, and states, which are used for tracking possibly entangled qubits. In GHZ (Figure 3c), each loop step in lines 4-6 entangles the qubit $x[j]$ with $x[0, j]$, i.e., the entangled qubit group is expanded from $x[0, j]$ to $x[0, j+1]$. However, the entanglement here is implicit: the program syntax does not directly tell if $x[j]$ is entangled with $x[0, j]$ but relies on an analysis to resolve the entanglement scopes, which is captured by introducing

- 1) *loci* (κ) to group possibly entangled qubits,
- 2) standard *kind environments* (Ω) to record variable *kinds* (explained below), and
- 3) *locus type environments* (σ) to keep track of both loci and their quantum state types.

QAFNY variables may represent one of three *kinds*² of values (Figure 4). C and M kinds are scalars; the former is an integer³, and the latter is a measurement outcome (r, n) where r is the probability of outcome n . Q m kind variables represent a physical m -length qubit array conceptually living in a heap. For simplicity, we assume no aliasing in variable names, no overlapping between qubit arrays referred to by any two different variables, and scalar and qubit array variables are always distinct. Quantum values are categorized into three different types: Nor, Had and EM. A *normal* value (Nor) is an array (tensor product) of single-qubit values $|0\rangle$ or $|1\rangle$. Sometimes, a (Nor)-typed value is associated with an amplitude z , representing an intermediate partial program state; an example is in Section 6.1. A *Hadamard* (Had) typed value represents a collection of qubits in superposition but not entangled, i.e., an n -qubit array $\frac{1}{\sqrt{2}}(|0\rangle + \alpha(r_0)|1\rangle) \otimes \dots \otimes \frac{1}{\sqrt{2}}(|0\rangle + \alpha(r_{n-1})|1\rangle)$, can be encoded as $\frac{1}{\sqrt{2^n}} \otimes_{j=0}^{n-1} (|0\rangle + \alpha(r_j)|1\rangle)$, with $\alpha(r_j) = e^{2\pi i r_j}$ ($r_j \in \mathbb{R}$) being the *local phase*, a special amplitude whose norm is 1, i.e., $|\alpha(r_j)| = 1$. The most general form of n -qubit values is the *entanglement* (EM) typed value, consisting of a linear combination (represented as an array) of basis-kets, as $\sum_{j=0}^m z_j \beta_j \eta_j$, where m is the number of elements in the array. In QAFNY, we extend traditional basis-ket structures in the Dirac notation to be the above form, so each basis-ket of the above value contains not only an amplitude z_j and a basis β_j but also a frozen basis stack η_j , storing bases not directly involved in the current computation. Here, β_j can always be represented as a single $|c_j\rangle$ by the equation in Figure 4. Every β_j in the array has the same cardinality, e.g., if $|c_0| = n$ ($\beta_0 = |c_0\rangle$), then $|c_i| = n$ ($\beta_j = |c_j\rangle$) for all j .

A QAFNY quantum state (φ), representing a quantum heap, maps *loci* to *quantum values*. Loci in a heap φ partition it into regions that contain possibly entangled qubits, with the guarantee that cross-locus qubits are not entangled. Each locus is a list of *disjoint ranges* (s), each represented by $x[n, m]$ – an in-place array slice selected from n to m (exclusive) in a physical qubit array x (always being Q kind). Ranges in a locus are pairwise disjoint, written as $s_1 \sqcup s_2$. For conciseness, we abbreviate a singleton range $x[j, j+1]$ as $x[j]$. At the type level, we maintain a locus type environment (σ) mapping loci to quantum types: any quantum state φ always has an entry in σ , guaranteeing that $\text{dom}(\varphi) = \text{dom}(\sigma)$, i.e., loci mentioned in φ and σ are the same. Locus type environments are stateful, i.e., a statement that starts with the environment σ could end up with a different one σ' because a locus could be modified during the execution. In addition to the locus type environment, we keep a kind environment between variables and their kinds. However, it is scoped and immutable as the kind of any scoped variable does not change.

² C and M kinds are also used as context modes in type checking. See Figure 9.

³ Any classical values are permitted in our implementation. For simplicity, we only consider integers here.

Basic Terms:	
Nat. Num $m, n \in \mathbb{N}$	Real $r \in \mathbb{R}$
Variable x, y	Bit $d ::= 0 \mid 1$
	Amplitude $z \in \mathbb{C}$
	Bitstring $c \in d^+$
	Phase $\alpha(r) ::= e^{2\pi ir}$
	Basis Vector $\beta ::= (c\rangle)^*$
Modes, Kinds, Types, and Classical/Quantum Values:	
Mode $g ::= \mathbb{C}$	\mathbb{M}
Classical Scalar Value $v ::= n$	(r, n)
Kind $g_k ::= g$	$\mathbb{Q} n$
Frozen Basis Stack $\gamma ::= (\hat{\beta})$	
Full Basis Vector $\eta ::= \beta\gamma$	
Basic Ket $w ::= z\eta$	
Quantum Type $\tau ::= \text{Nor}$	Had
Quantum Value (Forms) $q ::= w$	$\frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n-1} (0\rangle + \alpha(r_j) 1\rangle)$
Quantum Loci, Environment, and States	
Qubit Array Range $s ::= x[n, m)$	
Locus $\kappa ::= \bar{s}$	concatenated op \sqcup
Kind Environment $\Omega ::= x \rightarrow g_k$	
Type Environment $\sigma ::= \kappa : \tau$	concatenated op \sqcup
Quantum State (Heap) $\varphi ::= \bar{\kappa} : \bar{q}$	concatenated op \sqcup
Syntax Abbreviations and Basis/Locus Equations	
$\sum_{j=0}^0 w_j \simeq w_0$	$\sum_{j=0}^m w_j \simeq \sum_j w_j$
$1\gamma \simeq \gamma$	$z\beta(\emptyset) \simeq z\beta$
$ c_1\rangle c_2\rangle \equiv c_1 c_2\rangle$	$x[n, n) \equiv \emptyset$
$\emptyset \sqcup \kappa \equiv \kappa$	$x[n, m) \sqcup \kappa \equiv x[n, j) \sqcup x[j, m) \sqcup \kappa$ where $n \leq j \leq m$

Figure 4 QAFNY element syntax. Each range $x[n, m)$ in a locus represents the number range $[n, m)$ in physical qubit array x . Loci are finite lists, while type environments and states are finite sets. The operations after “concatenated op” are concatenations for loci, type environments, and quantum states.

On the bottom of Figure 4, we show abbreviations (\simeq) rules for presentation purposes; $A \simeq B$ means we write B to mean A . The left-most rule shows that **Nor** typed value is a singleton **EN** typed array; see the type-guided state rewrites in Section 3.3. We can also omit the (\emptyset) in a basis-ket presentation and color the basis stack with a hat sign $\widehat{\cdot}$, e.g., $\frac{1}{\sqrt{2}}|0\rangle \widehat{|1\rangle}$ means $\frac{1}{\sqrt{2}}|0\rangle (|1\rangle \mid)$; additionally, $\frac{1}{\sqrt{2}}|0\rangle \widehat{|1\rangle}$ means $\frac{1}{\sqrt{2}}|0\rangle |\widehat{1}\rangle (\emptyset)$. Below the \simeq rules in the figure, we present structural equations (\equiv) among bases and loci: 1) the locus concatenation \sqcup holds identity and associativity equational properties; 2) a range ($x[n, n)$) containing 0 qubit is empty; and 3) it is free to split a range ($x[n, m)$) into two ($x[n, j)$ and $x[j, m)$), preserving the disjointness of \sqcup .

3.2 Simultaneity for Tracking Qubit Positions and Entanglement Scopes

QAFNY uses locus transformations, captured by the type inference on program operations, to track entanglement scopes. Figure 5 describes the automated proof steps for verifying a loop-step in Figure 3c. The bottom pre-condition contains the quantum values for the two disjoint loci $x[0, j)$ and $x[j, n)$. The quantum conditional’s Boolean guard and body are applied to the qubits $x[j-1]$ and $x[j]$, respectively, appearing in the above two loci. The application entangles $x[j]$ with the locus $x[0, j)$ and transforms the locus to be $x[0, j+1)$, by appending $x[j]$ to the end of $x[0, j)$. The append, as the first (bottom) proof step in Figure 5, happens *automatically* through rewrites guided by the locus transformations in the type environment σ associated with each proof triple. After the rewrites, we preserve the property of no entangled cross-locus qubits.

In the above example, the static rewrites of a locus in a type environment simultaneously gear and change the rewrites of the locus value form in the associated state. We call this manipulation mechanism *simultaneity*. As shown in Section 1, quantum operations can apply on arbitrary qubit positions, which might seriously harm proof automation, based on previous experiments [16, 3] (Section 5.2), even if they tried hard for automation tactics. It is necessary to statically track qubit positions to permit the canonicalization of quantum state rewrites, allowing a uniform way of defining proof rules for operations.

$$\begin{array}{c}
 \frac{\Omega; \{\kappa_2 : \text{EN}\} \vdash_{\mathbb{M}} \left\{ \kappa_2 \mapsto \frac{1}{\sqrt{2}} |0\rangle |\bar{1}\rangle |\bar{1}\rangle \right\} e \left\{ \kappa_2 \mapsto \frac{1}{\sqrt{2}} |1\rangle |\bar{1}\rangle |\bar{1}\rangle \right\}}{\Omega; \{\kappa_1 : \text{EN}\} \vdash_{\mathbb{M}} \left\{ \kappa_1 \mapsto \frac{1}{\sqrt{2}} |\bar{1}\rangle |0\rangle |\bar{1}\rangle \right\} e \left\{ \kappa_1 \mapsto \frac{1}{\sqrt{2}} |\bar{1}\rangle |1\rangle |\bar{1}\rangle \right\}} \text{P-ORACLE} \\
 \frac{\Omega; \{\kappa_1 : \text{EN}\} \vdash_{\mathbb{M}} \left\{ \kappa_1 \mapsto \frac{1}{\sqrt{2}} |\bar{1}\rangle |0\rangle |\bar{1}\rangle \right\} e \left\{ \kappa_1 \mapsto \frac{1}{\sqrt{2}} |\bar{1}\rangle |1\rangle |\bar{1}\rangle \right\}}{\Omega; \{\kappa_1 : \text{EN}\} \vdash_{\mathbb{M}} \left\{ \mathcal{F}(x[j-1], \kappa_1) \mapsto \sum_{d=0}^1 \frac{1}{\sqrt{2}} |\bar{d}\rangle |0\rangle \right\} e \left\{ \kappa_1 \mapsto \frac{1}{\sqrt{2}} |\bar{1}\rangle |1\rangle |\bar{1}\rangle \right\}} \text{EQ} \\
 \frac{\Omega; \{\kappa_1 : \text{EN}\} \vdash_{\mathbb{M}} \left\{ \mathcal{F}(x[j-1], \kappa_1) \mapsto \sum_{d=0}^1 \frac{1}{\sqrt{2}} |\bar{d}\rangle |0\rangle \right\} e \left\{ \kappa_1 \mapsto \frac{1}{\sqrt{2}} |\bar{1}\rangle |1\rangle |\bar{1}\rangle \right\}}{\Omega; \{\kappa : \text{EN}\} \vdash_{\mathbb{C}} \left\{ \kappa \mapsto \sum_{d=0}^1 \frac{1}{\sqrt{2}} |\bar{d}\rangle |0\rangle \right\} \text{if } (x[j-1]) e \left\{ U(\neg x[j-1]) \mapsto \sum_{d=0}^1 \frac{1}{\sqrt{2}} |\bar{d}\rangle |0\rangle * U(x[j-1]) \mapsto \frac{1}{\sqrt{2}} |\bar{1}\rangle |1\rangle |\bar{1}\rangle \right\}} \text{M-F} \\
 \frac{\Omega; \{\kappa : \text{EN}\} \vdash_{\mathbb{C}} \left\{ \kappa \mapsto \sum_{d=0}^1 \frac{1}{\sqrt{2}} |\bar{d}\rangle |0\rangle \right\} \text{if } (x[j-1]) e \left\{ U(\neg x[j-1]) \mapsto \sum_{d=0}^1 \frac{1}{\sqrt{2}} |\bar{d}\rangle |0\rangle * U(x[j-1]) \mapsto \frac{1}{\sqrt{2}} |\bar{1}\rangle |1\rangle |\bar{1}\rangle \right\}}{\Omega; \sigma \vdash_{\mathbb{C}} \left\{ x[0, j] \mapsto \sum_{d=0}^1 \frac{1}{\sqrt{2}} |\bar{d}\rangle * x[j, n] \mapsto |\bar{0}\rangle \right\} \text{if } (x[j-1]) e \left\{ x[0, j+1] \mapsto \sum_{d=0}^1 \frac{1}{\sqrt{2}} |\bar{d}\rangle * x[j+1, n] \mapsto |\bar{0}\rangle \right\}} \text{P-IF} \\
 \kappa = x[j-1] \sqcup \kappa_1 \quad \kappa_1 = x[0, j-1] \sqcup x[j] \quad \kappa_2 = x[j] \sqcup x[0, j-1] \\
 e = x[j] \leftarrow x[j] + 1; \quad \sigma = \{x[0, j] : \text{EN}, x[j, n] : \text{Nor}\} \quad U(b) = u(b, x[j-1], \kappa)
 \end{array}$$

 **Figure 5** Detailed automated proof for a loop-step in GHZ. Constructed from the bottom up.

To permit automated proof inference, we design the uniformity in QAFNY proof rules to require that the locus fragments for qubits that an operation is directly applied always be prefixed. For example in Figure 5 P-IF, instead of having locus $x[0, j+1]$, we rewrite it further to $\kappa (x[j-1] \sqcup x[0, j-1] \sqcup x[j])$, so the qubit $x[j-1]$ that the Boolean guard is applied to appears at the start position. These rewrites simultaneously and appropriately rearrange the quantum value associated with the loci. In a QAFNY quantum state, qubits in a locus are arranged as a list of indices pointing to qubit positions. The locus indices point to particular qubits in a `Nor` and `Had` typed value since they essentially represent an array of qubits. An `EN` typed value consists of a list of basis-kets; the locus indices refer to the corresponding bases appearing in each basis-ket.

$$\boxed{x[j-1] \sqcup x[0, j-1] \sqcup x[j] \mapsto \sum_{d=0}^1 \frac{1}{\sqrt{2}} |\bar{d}\rangle |\bar{d}\rangle |0\rangle} \quad x[0, j-1] \sqcup x[j] \mapsto \frac{1}{\sqrt{2}} |\bar{1}\rangle |0\rangle |\bar{1}\rangle \quad x[j] \sqcup x[0, j-1] \mapsto \frac{1}{\sqrt{2}} |0\rangle |\bar{1}\rangle |\bar{1}\rangle$$

In the three example states above from Figure 5, the first maps the locus $\kappa (x[j-1] \sqcup x[0, j-1] \sqcup x[j])$ to the pre-state in the bottom of line P-IF, where $|\bar{d}\rangle$ is expanded to $|d\rangle |\bar{d}\rangle$. In each basis-ket (d is 0 or 1), the first qubit $x[j-1]$ of the locus κ corresponds to the first basis bit, while the last qubit $x[j]$ corresponds to $|0\rangle$, the last basis bit. Applying an operation on $x[j]$ performs the application on the last basis bit $|0\rangle$ for every basis-ket. We can also refer a consecutive fragment of a locus to its basis bits, e.g., range $x[0, j-1]$ refers to $|\bar{d}\rangle$, the middle portion of each basis-ket, provided that they have the same cardinality. In this paper, we call the corresponding basis bits of qubits or locus fragments for a value (or a basis-ket set) as the *qubit's/locus's position bases* of the value (or the basis-ket set). A locus's position bases are linked and moved according to the rewrites of the locus, e.g., the middle and right examples above represent the rewrites from locus $\kappa_1 = x[0, j-1] \sqcup x[j]$ to $\kappa_2 = x[j] \sqcup x[0, j-1]$ in the pre-states (bottom to upper) of Figure 5 line EQ.

The rewrite moves $x[j]$ in κ_1 to the front in κ_2 ; correspondingly, $x[j]$'s position basis ($|0\rangle$) is also moved to the front. These examples show the functionality of *frozen basis stacks*. The two basis-kets' frozen basis stacks both contain a basis $|\bar{1}\rangle$, which are not referenced by any part of a locus and therefore unreachable qubits. As shown in Section 1, we want local reasoning and preserving quantum theorems, i.e., a quantum state for a program piece does not mention qubits that are not reachable in the piece, e.g., accessing $x[j-1]$ above inside the conditional body means a violation of no-cloning. However, quantum states can be entangled, so unreachable qubits cannot be separated from the states. Instead, we hide

the unreachable qubits, such as $x[j-1]$, in the frozen stack and retrieve it after jumping out of the conditional body. A comprehensive example is given in Section 6.2. We show how to unfreeze the frozen bases and explain the motivation for having frozen bases shortly below.

3.3 Rewrites based on Locus Type and State Equivalence Relations

The QAFNY type system maintains simultaneity through the type-guided state rewrites, formalized as equivalence relations (Section 4.3). Other than the locus qubit position permutation introduced above, the types associated with loci in the environment also play an essential role in the rewrites. In QAFNY, a locus represents a possibly entangled qubit group. From the study of many quantum algorithms [2, 4, 35, 48, 1, 43, 30, 14], we found that the establishment of an entanglement group can be viewed as a loop structure of incrementally adding a qubit to the group at a time, representing the entanglement’s scope expansion; as the analogy in Figure 3b, qubits in the blue part are added to the orange part one by one. This behavior is similar to splits and joins of array elements if we view quantum states as arrays. However, joining and splitting two EN-typed values are hard problems ⁴. Another critical observation in studying many quantum algorithms is that the entanglement group establishment usually involves splitting a qubit in a `Nor/Had` typed value and joining it to an existing EN typed entanglement group. We manage these join and split patterns type-guided equations in QAFNY, suitable for automated verification. The GHZ example above (Figure 3c line 5) is an example of `Nor` and an EN type state split and join, where in each loop-step in Figure 3c, a `Nor`-typed qubit $x[j]$ is split from locus $x[j, n)$ and moved to the end of the EN-typed locus $x[0, j)$. Details are in Section 4.3.

3.4 The Qafny Proof System Glance Via Quantum Conditional Proofs

We integrate our type system with the QAFNY proof system, where QAFNY’s type-guided proof triple $(\Omega; \sigma \vdash_g \{P\} e \{Q\})$ states that from a pre-condition P , executing e results in a post-condition Q , provided that P and Q are resp. well-formed w.r.t σ and σ' , where $\Omega, \sigma \vdash_g e \triangleright \sigma'$ is a valid typing judgment (explained in Section 4.3).

A key design principle for proof automation rules is compositional and rule generalization, i.e., automated proof steps should be compositional, where each proof step is localized regarding a localized state, and the generalization means that automation should not depend on the specific local states. The issue with quantum proof rule designs is entanglement, i.e., a program execution on a local state might have global effects, which force the proof automation system to perform case analyses on the local states to resolve the global effects. For example, in verifying the conditional `if` ($x[j-1]$) e in the bottom of Figure 5, e can be applied to an entangled qubit state outside the visibility of qubits mentioned in e . Since e can be arbitrarily complicated, the prior work [56] handles the verification of the quantum conditional by expanding it as a whole matrix applied to a whole quantum state and performing case analyses. For proof automation, we need to design a uniform procedure, expressed as proof rules, to derive the verification; such a task is handled by predicate transformers and frozen stacks built on our locus structures.

An example is given at the line P-If in Figure 5, we utilize two locus predicate transformers \mathcal{F} and \mathcal{U} to transform the pre- and post-conditions so that they focus on the loci and basis-kets relevant to the current computation. In verifying the quantum conditional (`if` ($x[j-1]$) $\{x[j] \leftarrow x[j+1]\}$), we first apply \mathcal{F} to transform the pre-condition. For the value

⁴ The former is a Cartesian product; the latter is $\geq NP$ -hard, both unsuitable for automated verification.

$\frac{1}{\sqrt{2}} |\bar{0}\rangle |0\rangle + \frac{1}{\sqrt{2}} |\bar{1}\rangle |0\rangle$, we filter out the basis-ket $\frac{1}{\sqrt{2}} |\bar{0}\rangle |0\rangle$, as the Boolean guard ($x[j-1]$) is not satisfiable for $x[j-1]$'s position basis ($|0\rangle$) of the basis-ket. For the remaining basis-ket $\frac{1}{\sqrt{2}} |\bar{1}\rangle |0\rangle$, we freeze $x[j-1]$'s position basis ($|1\rangle$), by pushing $|1\rangle$ to the frozen stack as an unreachable position, highlighted by $\widehat{|1\rangle}$, since it represents the qubit appearing in the Boolean guard that should not join any computation in e ($x[j] \leftarrow x[j]+1$). A frozen stack represents the link between the local state and its entangled global state. Each basis-ket in a superposition state can be associated with a single frozen stack, and we utilize a predicate transformer to manipulate all these frozen stacks in a state, recording the side-effects of the entangled global state caused by local state changes.

Notice that the locus is transformed from κ to κ_1 by removing $x[j-1]$ to preserve simultaneity. The post-condition $\kappa_1 \mapsto \frac{1}{\sqrt{2}} |\bar{1}\rangle |1\rangle \widehat{|1\rangle}$ contains only the computation result of the basis-ket $\frac{1}{\sqrt{2}} |\bar{1}\rangle |0\rangle$, and we want the final post-state to contain all other missing pieces, which is the task of the two \mathcal{U} transformers. $\mathcal{U}(b, x[j-1], \kappa)$ points to the basis-ket satisfying the Boolean guard ($\frac{1}{\sqrt{2}} |\bar{1}\rangle |0\rangle$), from the above result post-condition. The transformer transforms $x[j-1]$'s position basis, currently in the stack, back to its normal position. $\mathcal{U}(\neg b, x[j-1], \kappa)$ represents the basis-ket not satisfying the guard ($\frac{1}{\sqrt{2}} |\bar{0}\rangle |0\rangle$), where we retrieve it from the pre-condition through the transformer. Finally, the two transformers transform and assemble the two states into one as the post-condition at the bottom Figure 5. Section 4.4 contains more details.

4 Qafny Formalism

This section formalizes QAFNY's syntax, semantics, type system, proof system, and the corresponding soundness and completeness theorems. Running example in Figure 6 describes quantum order finding, the core component of Shor's algorithm (complete one in TR [24] C.5). The program assumes that an n -qubit H gate and an addition ($y[0, n]+1$) respectively applied to ranges $x[0, n]$ and $y[0, n]$ before line 3. The for-loop entangles range $y[0, n]$ with every qubit in $x[0, n]$, one per loop step, and applies a modulo multiplication in each step. $\text{measure}(y)$ (partial measurement) in line 8 non-deterministically outputs a classical value $a^t \% N$ for y , and interconnectively rearranges $x[0, n]$'s quantum state, with all basis kets' bases related to a period value p . We unveil the details along with the section.

4.1 Qafny Syntax

QAFNY is a C-like flow-sensitive language equipped with quantum heap mutations, quantum conditionals, and for-loops. We intend to provide users with a high-level view of quantum operations, e.g., viewing H and $\text{QFT}^{[-1]}$ gates as state preparation, quantum oracles (μ in [23]) as quantum arithmetic operations, and controlled gates as quantum conditionals and loops. As in Figure 7, aside from standard forms such as sequence ($e ; e$) and $\text{SKIP}(\{\})$, statements e also include let binding ($\text{let } x = am \text{ in } e$), quantum heap mutations ($_ \leftarrow _$), quantum/classical conditionals ($\text{if } (b) e$), and loops ($\text{for } j \in [a_1, a_2] \&& b \{e\}$). The let statement binds either the result of an arithmetic expression (a) or a *computational basis measurement* operator ($\text{measure}(y)$) to an immutable C/M kind variable x in the body e . This design ensures all classical variables are immutable and lexically scoped to avoid quantum observer breakdown due to mutating a classical variable inside a quantum conditional body.

(1) `int u = 0; if (x[0]) u = 1;` ✗ (2) `if (x[0]) let u=1 in {};` ✓ (3) `let u=1 in if (x[0]) {};` ✓

Here, case (1) declares u as 0 and changes its value to 1 inside the quantum conditional,

```

 $1 < a < N \quad E(t) = x[t, n] \mapsto \frac{1}{\sqrt{2^n-t}} \bigotimes_{i=0}^{n-t-1} (|0\rangle + |1\rangle) * x[0, t] \boxplus y[0, n] \mapsto \sum_{i=0}^{2^t-1} \frac{1}{\sqrt{2^t}} |i\rangle |a^i \% N\rangle$ 
1 { $x[0, n] \mapsto \frac{1}{\sqrt{2^n}} \bigotimes_{j=0}^{n-1} (|0\rangle + |1\rangle) * y[0, n] \mapsto |\bar{0}\rangle |1\rangle$ } { $x[0, n] : \text{Had}, y[0, n] : \text{Nor}$ }
2 { $E(0)$ } { $x[0, n] : \text{Had}, x[0, 0] \boxplus y[0, n] : \text{EN}$ }
3 for  $j \in [0, n)$  &&  $x[j]$  { $x[j, n] : \text{Had}, x[0, j] \boxplus y[0, n] : \text{EN}$ }
4 { $E(j)$ } { $x[0, 0] : \text{Had}, x[0, n] \boxplus y[0, n] : \text{EN}$ }
5  $y[0, n] \leftarrow a^{2^j} \cdot y[0, n] \% N;$  { $x[0, 0] : \text{Had}, x[0, n] \boxplus y[0, n] : \text{EN}$ }
6 { $E(n)$ } { $x[0, n] \boxplus y[0, n] \mapsto \sum_{i=0}^{2^n-1} \frac{1}{\sqrt{2^n}} |i\rangle |a^i \% N\rangle$ } { $x[0, n] \boxplus y[0, n] : \text{EN}$ }
7 let  $u = \text{measure}(y)$  in ...
8 { $x[0, n] \mapsto \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} |t+kp\rangle * p = \text{ord}(a, N)$ } { $x[0, n] : \text{EN}$ }
9 { $* u = (\frac{p}{2^n}, a^t \% N) * r = \text{rnd}(\frac{2^n}{p})$ } { $x[0, n] : \text{EN}$ }

```

Figure 6 Snippets from quantum order finding in Shor’s algorithm; full proof in TR [24] C.5. $\text{ord}(a, N)$ gets the order of a and N . $\text{rnd}(r)$ rounds r to the nearest integer. We interpret integers as bitstrings in $|i\rangle$ and $|a^i \% N\rangle$. The right column presents the types of all loci involved.

OQASM Expr	μ					
Arith Expr	$a ::= x$	$ x[i, j)$	$ v$	$ a_1 + a_2$	$ a_1 \cdot a_2$	$ \dots$
Bool Expr	$b ::= x[a]$	$ (a_1 = a_2) @ x[a]$		$ (a_1 < a_2) @ x[a]$		$ \dots$
Predicate Locus	$K ::= \kappa$	$ \mathcal{M}(x, n, \kappa)$	$ \mathcal{F}(b, \kappa, \kappa)$	$ \mathcal{U}(b, \kappa, \kappa)$		
Predicate	$P, Q, R ::= a_1 = a_2$	$ a_1 < a_2$	$ K \mapsto q$	$ P \wedge P$	$ P * P$	$ \dots$
Gate Expr	$op ::= \text{H}$	$ \text{QFT}^{[-1]}$				
C/M Kind Expr	$am ::= a$	$ \text{measure}(y)$				
Statement	$e ::= \{\}$	$ \kappa \leftarrow op$	$ \kappa \leftarrow \mu$	$ \text{let } x = am \text{ in } e$		
		$ e_1 ; e_2$	$ \text{if } (b) e$		$ \text{for } j \in [a_1, a_2) \&& b \{e\}$	

Figure 7 Core QAFNY syntax. Element syntax is in Figure 4 and oQASM is in [23]. $\text{QFT}^{[-1]}$ refers to the QFT and reversed QFT. An arithmetic expression x is a C/M kind variable, $x[i, j)$ is a quantum array range, and v is a C/M kind value. $x[a]$ is the a -th element of qubit array x .

which creates an observer effect because u ’s value depends on qubit $x[0]$. Cases (2) and (3) show that our immutable `let` binding can avoid the issue because the binding in (2) can be compiled to (3) due to the immutability; thus, u ’s value does not depend on the qubit.

A quantum heap mutation operation mutates qubit array data by applying to a locus κ either a unitary state preparation operation (op) (one of Hadamard H , quantum Fourier transformation QFT , and its inverse QFT^{-1}) or a unitary oracle operation (μ).⁵ Other unitary operations, including quantum diffusion and amplification operations, are in TR [24] C.1.

Quantum reversible Boolean guards b are implemented as OQASM oracle operations, expressed by one of $(a_1 = a_2) @ x[a]$, $(a_1 < a_2) @ x[a]$, and $x[a]$, which intuitively amounts to computing $a_1 = a_2$, $a_1 < a_2$ and `false` respectively as b_0 and storing the result of $b_0 \oplus x[a]$ as a binary in qubit $x[a]$.⁶ In both conditionals and loops, guards b are used to represent the qubits that are controlling. In addition to the `let` bindings, the quantum for-loop also introduces and enumerates C-kind value j over interval $[a_1, a_2)$ with j visible to both the guard b and the loop body e . As a result of immutability, loops in QAFNY are guaranteed to terminate. If all variables in the guard b are classical, the conditional or

⁵ μ can define all quantum arithmetics, e.g., $x[j]+1$ (Fig. 3c) & $a^{2^j} y[0, n] \% N$ (Fig. 6). See [23].

⁶ a_1 and a_2 can possibly apply to a range, like $y[0, n)$, in an entangled locus.

$$\begin{array}{c}
 \text{S-EXPC} \quad \frac{(\varphi, e[n/x]) \Downarrow \varphi'}{(\varphi, \text{let } x = n \text{ in } e) \Downarrow \varphi'} \quad \text{S-EXPM} \quad \frac{(\varphi, e[(r, n)/x]) \Downarrow \varphi'}{(\varphi, \text{let } x = (r, n) \text{ in } e) \Downarrow \varphi'} \quad \text{S-OP} \\
 \frac{}{(\varphi \uplus \{\kappa \boxdot \kappa' : q\}, \kappa \leftarrow \circ) \Downarrow \varphi \uplus \{\kappa \boxdot \kappa' : \llbracket \circ \rrbracket^{|\kappa|} q\}} \quad \circ = op \vee \circ = \mu
 \\[10pt]
 \text{S-IF} \quad \frac{FV(\Omega, b) = \kappa \quad \llbracket b \rrbracket^{|\kappa|} q = q(\kappa, b) + q(\kappa, \neg b) \quad (\varphi \uplus \{\kappa' : S^{|\kappa|}(q(\kappa, b))\}, e) \Downarrow \varphi \uplus \{\kappa' : q'\}}{(\varphi \uplus \{\kappa \boxdot \kappa' : q\}, \text{if } (b) e) \Downarrow \varphi \uplus \{\kappa \boxdot \kappa' : P(q') + q(\kappa, \neg b)\}}
 \\[10pt]
 \text{S-LOOP} \quad \frac{n < n' \quad (\varphi, \text{if } (b[n_1/j]) e[n_1/j]) \Downarrow \varphi' \quad (\varphi', \text{for } j \in [n+1, n') \And b \{e\}) \Downarrow \varphi''}{(\varphi, \text{for } j \in [n, n') \And b \{e\}) \Downarrow \varphi''} \quad \text{S-LOOP1} \quad \frac{n \geq n'}{(\varphi, \text{for } j \in [n, n') \And b \{e\}) \Downarrow \varphi}
 \\[10pt]
 \text{S-SEQ} \quad \frac{(\varphi, e_1) \Downarrow \varphi' \quad (\varphi', e_2) \Downarrow \varphi''}{(\varphi, e_1 ; e_2) \Downarrow \varphi''} \quad \text{S-MEA} \quad \frac{\kappa = y[0, n] \quad r = \sum_j |z_j|^2 \quad (\varphi \uplus \{\kappa' : \sum_j \frac{z_j}{\sqrt{r}} \eta_j\}, e[(r, \{c\})/x]) \Downarrow \varphi'}{(\varphi \uplus \{\kappa \boxdot \kappa' : \sum_j z_j |c\rangle \eta_j + q(\kappa, c \neq \kappa)\}, \text{let } x = \text{measure}(y) \text{ in } e) \Downarrow \varphi'}
 \\[10pt]
 \begin{array}{lcl}
 \llbracket \circ \rrbracket^n (\sum_j z_j |c_j\rangle \eta_j) & \triangleq & \sum_j z_j (\llbracket \circ \rrbracket |c_j\rangle) \eta_j \quad \text{where } (\circ = \mu \vee \circ = op \vee \circ = b) \wedge \forall j |c_j| = n \\
 (\sum_i z_i |c_i\rangle \eta_i + q)(\kappa, b) & \triangleq & \sum_i z_i |c_i\rangle \eta_i \quad \text{where } \forall i |c_i| = |\kappa| \wedge \llbracket b[c_i/\kappa] \rrbracket = \text{true} \\
 S^n (\sum_j z_j |c_j\rangle \beta_j (\beta'_j)) & \triangleq & \sum_j z_j \beta_j (\beta'_j) \quad \text{where } \forall j |c_j| = n \\
 P(\sum_j z_j \beta_j (\beta'_j)) & \triangleq & \sum_j z_j |c_j\rangle \beta_j (\beta'_j)
 \end{array}
 \end{array}$$

 **Figure 8** Selected semantic rules. $\{c\}$ turns basis c to an integer.

loop becomes a standard classical one, which is differentiated and definable by our type system, described in TR [24] C. Obviously, users can always view a QAFNY program as a quantum sub-component in a Dafny program, which provides better library support for classical conditionals. Predicates and predicate loci in Figure 7 describe quantum state properties in the QAFNY proof system, explained in Section 4.4.

4.2 Qafny Semantics

The QAFNY semantics is formalized as a big-step transition relation $(\varphi, e) \Downarrow \varphi'$, with φ / φ' being quantum states as described in Figure 4. The judgment relation states that a program e with the pre-state φ transitions to a post-state φ' . A selection of the rules defining \Downarrow may be found Figure 8, and the additional rules are in TR [24] C.2. $FV(\Omega, -)$ produces a locus by unioning all qubits in $-$ with the quantum variable kind information in Ω ; its definition is given in TR [24] A.

Assignment and Mutation Operations. Rules S-EXPC and S-EXPM define the behaviors for C and M kind classical variable assignments, which perform variable-value substitutions. Rule S-OP defines a quantum heap mutation applying a state preparation operation (op) or an oracle expression (μ) to a locus κ for a EN-typed state. Here, the locus fragment κ to which the operation is applied must be the very first one in the locus $\kappa \boxdot \kappa'$ that refers to the entire quantum state q . If not, we will first apply equivalence rewrites to be explained in Section 4.3 to move κ to the front. With κ preceding the rest fragment κ' , the operation’s semantic function $\llbracket \circ \rrbracket^n$ (\circ being H or μ) is then applied to κ ’s position bases in the quantum value q . More specifically, the function is only applied to the first n (equal to $|\kappa|$) basis bits of each basis-ket in the value while leaving the rest unchanged. The semantic interpretations of the op and μ operations are essentially the quantum gate semantics given in Li et al. [23]. For example, in Figure 5 line P-ORACLE, we apply an oracle operation $x[j]+1$ to $x[j]$, the first position of the locus κ_2 (i.e., $x[j] \boxdot x[0, j-1]$), which transforms the first basis bit to $|1\rangle$.

Before the application, we rewrite the pre-state containing κ_1 below the line EQ in Figure 5, to the form corresponding to the locus κ_2 above the line.

Quantum Conditionals. As in rule S-IF, for a conditional $\text{if } (b) e$, we first evaluate the Boolean guard b on κ 's position bases ($FV(\Omega, b) = \kappa$) of the quantum value state q to $\llbracket b \rrbracket^{|\kappa|} q$ ⁷ because b 's computation might have side-effects in changing κ 's position bases, as the example in Section 6.2. The quantum value referred by $\kappa \sqcup \kappa'$ is further partitioned into $q\langle\kappa, b\rangle + q\langle\kappa, \neg b\rangle$ where $q\langle\kappa, b\rangle$ is a set of basis-kets whose κ 's position bases satisfying b and $q\langle\kappa, \neg b\rangle$ is the rest. Since the body e only affects the basis-kets ($q\langle\kappa, b\rangle$) satisfying the guard b , we rule out the basis-kets $q\langle\kappa, \neg b\rangle$ (unsatisfying the guard) in e 's computation. We also need to push κ 's position bases in $q\langle\kappa, b\rangle$ to the frozen stacks through the S^n operation to maintain the locus-state simultaneity in Section 3.2.

We describe rule S-IF along with an example in Figure 6 line 3-5. Here, the j -th iteration is unrolled to a quantum conditional $\text{if } (x[j]) \{y[0, n] \leftarrow a^{2^j} \cdot y[0, n]\% N\}$. The loci involved in the computation are $x[j]$ and $x[0, j] \sqcup y[0, n]$, and their state transitions are given as:

$$\begin{aligned} & \left\{ x[j] : \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \right\} \sqcup \left\{ x[0, j] \sqcup y[0, n] : \sum_{i=0}^{2^j-1} \frac{1}{\sqrt{2^{j+1}}} |i\rangle |a^i \% N\rangle \right\} \\ \equiv & \left\{ x[j] \sqcup x[0, j] \sqcup y[0, n] : \sum_{i=0}^{2^j-1} \frac{1}{\sqrt{2^{j+1}}} |1\rangle |i\rangle |a^i \% N\rangle + \sum_{i=0}^{2^j-1} \frac{1}{\sqrt{2^{j+1}}} |0\rangle |i\rangle |a^i \% N\rangle \right\} \\ \xrightarrow{\text{S-IF}} & \left\{ x[j] \sqcup x[0, j] \sqcup y[0, n] : \sum_{i=0}^{2^j-1} \frac{1}{\sqrt{2^{j+1}}} |1\rangle |i\rangle |(a^i \cdot a^{2^j}) \% N\rangle + \sum_{i=0}^{2^j-1} \frac{1}{\sqrt{2^{j+1}}} |0\rangle |i\rangle |a^i \% N\rangle \right\} \\ \equiv & \left\{ x[0, j+1] \sqcup y[0, n] : \sum_{i=0}^{2^{j+1}-1} \frac{1}{\sqrt{2^{j+1}}} |i\rangle |a^i \% N\rangle \right\} \end{aligned}$$

The first equation transition (\equiv) merges the two locus states and turns the merged state into two sets (separated by \sqcup), respectively representing basis-kets where $x[j]$'s position bases are 1 and 0. Since the Boolean guard $x[j]$ has no side-effects, the application $\llbracket b \rrbracket^{|\kappa|}$ is an identity. The S-IF application performs a modulo multiplication oracle application on the basis-ket set where $x[j]$'s position bases being 1, while the last equation merges the two sets back to one summation formula. The S-IF application above can be further decomposed into two additional transitions in between:

$$\begin{aligned} \longrightarrow & \left\{ x[0, j] \sqcup y[0, n] : \sum_{i=0}^{2^j-1} \frac{1}{\sqrt{2^{j+1}}} |i\rangle |a^i \% N\rangle |\overline{1}\rangle \right\} \\ \xrightarrow{\text{S-OP}} & \left\{ x[0, j] \sqcup y[0, n] : \sum_{i=0}^{2^j-1} \frac{1}{\sqrt{2^{j+1}}} |i\rangle |(a^i \cdot a^{2^j}) \% N\rangle |\overline{1}\rangle \right\} \end{aligned}$$

The first transition removes the $q\langle\kappa, \neg b\rangle$ part, e.g., the basis-ket set where $x[j]$'s position bases are 0. Additionally, for every basis-ket in the $q\langle\kappa, b\rangle$ set, e.g., the basis-ket set where $x[j]$'s position bases being 1, we freeze κ 's position bases by pushing the bases into the basis-ket's stacks through the function application $S^{|\kappa|}(q\langle\kappa, b\rangle)$, which finds the first $|\kappa|$ bits in every basis-ket and push them into the basis-ket's stack so that e 's application targets locus κ' instead of $\kappa \sqcup \kappa'$. As the first transition above, for each basis-ket, we push $x[j]$'s position basis ($|1\rangle$) to the basis-ket's stack, as the $|\overline{1}\rangle$ part and the pointed-to locus is rewritten to $x[0, j] \sqcup y[0, n]$. After applying the body e to the state, for every basis-ket, we pop κ 's position bases ($P(q')$) from the basis-ket's stack and relabel the locus of the state to be $\kappa \sqcup \kappa'$; in doing so, we also need to add the unmodified basis-kets $q\langle\kappa, \neg b\rangle$ back into the whole state. After applying S-OP on locus $x[0, j] \sqcup y[0, n]$ above, we pop $|\overline{1}\rangle$ from every basis-ket's stack and assemble the unchanged part ($\sum_{i=0}^{2^j-1} \frac{1}{\sqrt{2^{j+1}}} |0\rangle |i\rangle |a^i \% N\rangle$) back to the state of locus $x[j] \sqcup x[0, j] \sqcup y[0, n]$; the result is shown as the state after the $\xrightarrow{\text{S-IF}}$ application above.

⁷ This is defined formally as an oracle, same as μ above.

$$\begin{array}{c}
 \text{T-PAR} \quad \frac{\sigma \preceq \sigma' \quad \Omega; \sigma' \vdash_g e \triangleright \sigma''}{\Omega; \sigma \vdash_g e \triangleright \sigma''} \quad \text{T-EXPC} \quad \frac{x \notin \text{dom}(\Omega) \quad \Omega; \sigma \vdash_g e[n/x] \triangleright \sigma'}{\Omega; \sigma \vdash_g \text{let } x = n \text{ in } e \triangleright \sigma'} \quad \text{T-EXPM} \quad \frac{x \notin \text{dom}(\Omega) \quad \Omega \vdash a : \mathbb{M} \quad \Omega[x \mapsto \mathbb{M}]; \sigma \vdash_g e \triangleright \sigma'}{\Omega; \sigma \vdash_g \text{let } x = a \text{ in } e \triangleright \sigma'} \\
 \text{T-OP} \quad \frac{\circ = op \vee \circ = \mu}{\Omega; \sigma \uplus \{\kappa \sqsubseteq \kappa' : \text{EN}\} \vdash_g \kappa \leftarrow \circ \triangleright \sigma \uplus \{\kappa \sqsubseteq \kappa' : \text{EN}\}} \quad \text{T-MEA} \quad \frac{\Omega(y) = \mathbb{Q} \ n \quad x \notin \text{dom}(\Omega) \quad \Omega[x \mapsto \mathbb{M}]; \sigma \uplus \{\kappa : \text{EN}\} \vdash_c e \triangleright \sigma'}{\Omega; \sigma \uplus \{y[0, n) \sqsubseteq \kappa : \tau\} \vdash_c \text{let } x = \text{measure}(y) \text{ in } e \triangleright \sigma'} \\
 \text{T-IF} \quad \frac{FV(\Omega, b) = \kappa \quad FV(\Omega, e) \subseteq \kappa' \quad \Omega; \sigma \uplus \{\kappa' : \text{EN}\} \vdash_{\mathbb{M}} e \triangleright \sigma \uplus \{\kappa' : \text{EN}\}}{\Omega; \sigma \uplus \{\kappa \sqsubseteq \kappa' : \text{EN}\} \vdash_g \text{if } (b) \ e \triangleright \sigma \uplus \{\kappa \sqsubseteq \kappa' : \text{EN}\}} \quad \text{T-SEQ} \quad \frac{\Omega; \sigma \vdash_g e_1 \triangleright \sigma_1 \quad \Omega; \sigma_1 \vdash_g e_2 \triangleright \sigma_2}{\Omega; \sigma \vdash_g e_1 ; e_2 \triangleright \sigma_2} \\
 \text{T-LOOP} \quad \frac{x \notin \text{dom}(\Omega) \quad \forall j \in [n_1, n_2]. \Omega; \sigma[j/x] \vdash_g \text{if } (b[j/x]) \ e[j/x] \triangleright \sigma[j+1/x]}{\Omega; \sigma[n_1/x] \vdash_g \text{for } x \in [n_1, n_2] \ \&& \ b \{e\} \triangleright \sigma[n_2/x]}
 \end{array}$$

 **Figure 9** QAFNY type system. $FV(\Omega, -)$ gets a locus containing qubits in $-$ w.r.t. Ω (TR [24] A).

Quantum Measurement. A measurement (`let x = measure(y) in e`) collapses a qubit array y , binds a \mathbb{M} -kind outcome to x , and restricts its usage in e . Rule S-MEA shows the partial measurement behavior⁸. Assume that the locus containing the qubit array y is $y[0, n) \sqsubseteq \kappa'$, the measurement is essentially a two-step array filter: (1) the basis-kets of the EN typed value is partitioned into two sets (separated by $+$): $(\sum_{j=0}^m z_j |c\rangle |c_j\rangle) + q\langle \kappa, c \neq \kappa \rangle$ with $\kappa = y[0, n)$, by randomly picking a $|\kappa|$ -length basis c where every basis-ket in the first set have κ 's position basis c ; and (2) we create a new array value by removing all the basis-kets not having c as prefixes (the $q\langle \kappa, c \neq \kappa \rangle$ part) and also removing the κ 's position basis in every remaining basis-ket; thus, the quantum value becomes $\sum_{j=0}^m \frac{z_j}{\sqrt{r}} \eta_j$. Notice that the element size of the post-state $m+1$ is smaller than the size of the pre-state before the measurement. Since the amplitudes of basis-kets must satisfy $\sum_i |z_i|^2 = 1$, we need to normalize the amplitude of each element in the post-state by multiplying a factor $\frac{1}{\sqrt{r}}$, with $r = \sum_{j=0}^m |z_j|^2$ as the sum of the amplitude squares appearing in the post-state. When proving quantum program properties, the amplitudes appearing in basis-kets usually follow a periodic pattern that users can provide, so computing r will be relatively simple, see Section 4.4. In Figure 6, the measurement (line 8) transitions the state from lines 7 to 9. Locus $y[0, n)$'s position basis is $|a^i \% N\rangle$ for each basis-ket in $\sum_{i=0}^{2^n-1} \frac{1}{\sqrt{2^n}} |i\rangle |a^i \% N\rangle$. We then randomly pick the basis value $a^t \% N$ as a measurement result, stored in u , and the probability of the pick is $\frac{p}{2^n}$ where p is the order of a and N . The probability is computed solely based on p because it represents the period of the factorization in Shor's algorithm. The number (r) of remaining basis-kets in range $x[0, n)$ is computed by rounding $\frac{2^n}{p}$.

4.3 Qafny Locus Type System

The QAFNY typing judgment $\Omega; \sigma \vdash_g e \triangleright \sigma'$ states that e is well-typed under the context mode g (the syntax of kind g is reused as context modes) and environments Ω and σ . The kind environment Ω is populated through `let` and `for` loops that introduce \mathbb{C} and \mathbb{M} kind variables, while \mathbb{Q} -kind variable mappings in Ω are given as a global environment. Selected type rules are in Figure 9; the rules not mentioned are similar and given in TR [24] C.3. For every type rule, well-formed domains ($\Omega \vdash \text{dom}(\sigma)$) are required but hidden from the rules,

⁸ A complete measurement is a special case of a partial measurement when κ' is empty in S-MEA

$$\begin{array}{ll}
\sigma \preceq \sigma & \varphi \equiv \varphi \\
\{\emptyset : \tau\} \uplus \sigma \preceq \sigma & \{\emptyset : q\} \uplus \varphi \equiv \varphi \\
\{\kappa : \tau\} \uplus \sigma \preceq \{\kappa : \tau'\} \uplus \sigma & \{\kappa : q\} \uplus \varphi \equiv \{\kappa : q'\} \uplus \varphi \\
\text{where } \tau \sqsubseteq \tau' & \text{where } q \equiv_{|\kappa|} q' \\
\{\kappa_1 \sqcup s_1 \sqcup s_2 \sqcup \kappa_2 : \tau\} \uplus \sigma \preceq \{\kappa_1 \sqcup s_2 \sqcup s_1 \sqcup \kappa_2 : \tau\} \uplus \sigma & \{\kappa_1 \sqcup s_1 \sqcup s_2 \sqcup \kappa_2 : q\} \uplus \varphi \equiv \{\kappa_1 \sqcup s_2 \sqcup s_1 \sqcup \kappa_2 : q'\} \uplus \varphi \\
& \text{where } q' = q^{|\kappa_1|} (|s_1| \asymp |s_2|) \\
\{\kappa_1 : \tau\} \uplus \{\kappa_2 : \tau\} \uplus \sigma \preceq \{\kappa_1 \sqcup \kappa_2 : \tau\} \uplus \sigma & \{\kappa_1 : q_1\} \uplus \{\kappa_2 : q_2\} \uplus \varphi \equiv \{\kappa_1 \sqcup \kappa_2 : q'\} \uplus \varphi \\
& \text{where } q' = q_1 \bowtie q_2 \\
\{\kappa_1 \sqcup \kappa_2 : \tau\} \uplus \sigma \preceq \{\kappa_1 : \tau\} \uplus \{\kappa_2 : \tau\} \uplus \sigma & \{\kappa_1 \sqcup \kappa_2 : \varphi\} \uplus \sigma \equiv \{\kappa_1 : \varphi_1\} \uplus \{\kappa_2 : \varphi_2\} \uplus \sigma \\
& \text{where } \varphi_1 \bowtie \varphi_2 = \varphi \wedge |\varphi_1| = |\kappa_1|
\end{array}$$

(a) Environment Equivalence.

(b) State Equivalence.

Permutation:

$$(q_1 \otimes q_2 \otimes q_3 \otimes q_4)^n \langle i \asymp k \rangle \triangleq q_1 \otimes q_3 \otimes q_2 \otimes q_4 \quad \text{where } |q_1| = n \wedge |q_2| = i \wedge |q_3| = k$$

$$(\sum_j z_j |c_j\rangle |c'_j\rangle |c''_j\rangle \eta_j)^n \langle i \asymp k \rangle \triangleq \sum_j z_j |c_j\rangle |c''_j\rangle |c'_j\rangle \eta_j \quad \text{where } |c_j| = n \wedge |c'_j| = i \wedge |c''_j| = k$$

Join Product:

$$z_1 |c_1\rangle \bowtie z_2 |c_2\rangle \triangleq (z_1 \cdot z_2) |c_1\rangle |c_2\rangle \quad \sum_{j=0}^n z_j |c_j\rangle \bowtie \sum_{k=0}^m z_k |c_k\rangle \triangleq \sum_{j=0}^{n+m} z_j \cdot z_k |c_j\rangle |c_k\rangle$$

$$|c_1\rangle \bowtie \sum_j z_j \eta_j \triangleq \sum_j z_j |c_1\rangle \eta_j \quad (|0\rangle + \alpha(r) |1\rangle) \bowtie \sum_j z_j \eta_j \triangleq \sum_j z_j |0\rangle \eta_j + \sum_j (\alpha(r) \cdot z_j) |1\rangle \eta_j$$

Figure 10 QAFNY type/state relations. \cdot is math mult. Term $\sum^{n+m} P$ is a summation omitting the indexing details. \otimes expands a Had array, as $\frac{1}{\sqrt{2^{n+m}}} \otimes_{j=0}^{n+m-2} q_j = (\frac{1}{\sqrt{2^n}} \otimes_{j=0}^{n-1} q_j) \otimes (\frac{1}{\sqrt{2^m}} \otimes_{j=0}^{m-1} q_j)$.

such that every variable used in all loci of σ must appear in Ω , while $\Omega \vdash a : \mathbb{M}$ judges that the expression a is well-formed and returns an \mathbb{M} kind; see TR [24] A and B. The type system enforces three properties below.

No Cloning and Observer Breakdown. We enforce no cloning by disjointing qubits mentioned in a quantum conditional Boolean guard and its body. In rule T-IF, κ and κ' are disjoint unioned, and the two *FV* side-conditions ensure that the qubits mentioned in the Boolean guard and conditional body are respectively within κ and κ' ; thus, they do not overlap. QAFNY is a *flow-sensitive* language, as we enforce no observer breakdown by ensuring no classical variable assignments through the QAFNY syntax and no measurements inside a quantum conditional through context restrictions. Each program begins with the context mode C , which permits all QAFNY operations. Once a type rule switches the mode to M , as in T-IF, measurement operations are suspended in this scope, as T-MEA is valid only if the context mode is C . For instance, let's imagine that the measurement in Figure 6 line 8 lives inside the for-loop in line 5, which our type system would forbid because type checking through T-LOOP calls rule T-IF that marks the context mode to M , while the application of rule T-MEA requires a C mode context to begin with.

Guiding Locus Equivalence and Rewriting. The semantics in Section 4.2 assumes that the loci in quantum states can be in ideal forms, e.g., rule S-OP assumes that the target locus κ are always prefixed. This step is valid if we can rewrite (type environment partial order \preceq) the locus to the ideal form through rule T-PAR, which interconnectively rewrites the locus appearing in the state, through our state equivalence relation (\equiv), as the locus state simultaneity enforcement (Section 3.2). The state equivalence rewrites have two components.

First, the type and quantum value forms have simultaneity, i.e., given a type τ_1 for a locus κ in a type environment (σ), if it is a subtype (\sqsubseteq) of another type τ_2 , κ 's value q_1 in a state (φ) can be rewritten to q_2 that has the type τ_2 through state equivalence rewrites (\equiv_n) where n is the number of qubits in q_1 and q_2 . Both \sqsubseteq and \equiv_n are reflexive and

types `Nor` and `Had` are subtypes of `EN`, which means that a `Nor` typed value ($|c\rangle$) and a `Had` typed value ($\frac{1}{\sqrt{2^n}} \otimes_{j=0}^{n-1} (|0\rangle + \alpha(r_j)|1\rangle)$) can be rewritten to an `EN` typed value (TR [24] C.3). For example, range $x[0, n]$'s `Had` typed value $\frac{1}{\sqrt{2^n}} \otimes_{j=0}^{n-1} (|0\rangle + |1\rangle)$ in Figure 6 line 1 can be rewritten to an `EN` type as $\sum_{i=0}^{2^n-1} \frac{1}{\sqrt{2^n}} |i\rangle$. If such a rewrite happens, we correspondingly transform $x[0, n]$'s type to `EN` in the type environment.

Second, type environment partial order (\preceq) and state equivalence (\equiv) also have simultaneity – in a proof judgment, we associate the state predicate, representing a state φ , with the type environment σ by sharing the same domain, i.e., $\text{dom}(\varphi) = \text{dom}(\sigma)$. Thus, the environment rewrites (\preceq) happening in σ gear the state rewrites in φ , e.g., the bottom proof step of Figure 5 transforms locus $x[0, j]$ in σ to locus κ ($x[j-1] \sqcup x[0, j-1] \sqcup x[j]$) above it, and the state rewrites in the pre-condition predicate happen accordingly as (left to right):

$$\begin{array}{lll} \{x[0, j] : \sum_{d=0}^1 \frac{1}{\sqrt{2}} |\bar{d}\rangle\} \uplus \{x[j] : |0\rangle\} & \equiv & \{x[0, j+1] : \sum_{d=0}^1 \frac{1}{\sqrt{2}} |\bar{d}\rangle |0\rangle\} \\ \{x[0, j] : \text{EN}\} & \uplus \{x[j] : \text{Nor}\} & \preceq \{x[0, j+1] : \text{EN}\} \\ & & \preceq \{\kappa : \sum_{d=0}^1 \frac{1}{\sqrt{2}} |\bar{d}\rangle |0\rangle\} \end{array}$$

Here, we add qubit $x[j]$ ($|0\rangle$) to the end of locus $x[0, j]$ and transform locus $x[0, j+1]$ to κ , so the upper proof step (P-IF) in Figure 5 can proceed. The above rewrites are derived by the rules in Figure 10, where the rules in environment partial order and state equivalence are one-to-one corresponding. The first three lines describe the properties of reflective, identity, and subtyping equivalence. The fourth line enforces that the environment and state are close under locus permutation. After the equivalence rewrite, the position bases of ranges s_1 and s_2 are mutated by applying the function $q^{|\kappa_1|}(|s_1| \asymp |s_2|)$. One example is the locus rewrite in Figure 6 line 7 from left to right, as:

$$\begin{array}{lll} \{x[0, n] \sqcup y[0, n] : \text{EN}\} & \preceq & \{y[0, n] \sqcup x[0, n] : \text{EN}\} \\ \{x[0, n] \sqcup y[0, n] : \sum_{i=0}^{2^n-1} \frac{1}{\sqrt{2^n}} |i\rangle |a^i \% N\rangle\} & \equiv & \{y[0, n] \sqcup x[0, n] : \sum_{i=0}^{2^n-1} \frac{1}{\sqrt{2^n}} |a^i \% N\rangle |i\rangle\} \end{array}$$

The last two lines in Figures 10a and 10b describe locus joins and splits, where the latter is an inverse of the former but much harder to perform practically. In the most general form, joining two `EN`-typed states computes the Cartesian product of their basis-kets, shown in the bottom of Figure 10, which is practically hard for proof automation. Fortunately, the join operations in most quantum algorithms are between a `Nor/Had` typed and an `EN`-typed state. Joining a `Nor`-typed and `EN`-typed state puts extra qubits in the right location in every basis-ket of the `EN`-typed state as discussed in Section 3.3. Joining a `Had`-typed qubit (single qubit state) and `EN`-typed state duplicates the `EN`-typed basis-kets. In every loop step in Figure 6 line 3-5, we add a `Had`-typed qubit $x[j]$ to the middle of an `EN`-typed locus $x[0, j] \sqcup y[0, n]$, transform the state to:

$$\{x[0, j+1] \sqcup y[0, n] : \sum_{i=0}^{2^j-1} \frac{1}{\sqrt{2^j}} |i\rangle |0\rangle |a^i \% N\rangle + \sum_{j=0}^{2^j} \frac{1}{\sqrt{2^{j-1}}} |i\rangle |1\rangle |a^i \% N\rangle\}$$

The state can be further rewritten to the one in Figure 6 by merging the above two parts (separated by $+$). Notice that the basis-kets are still all distinct because the two parts are distinguished by $x[j]$'s position basis, i.e., $|0\rangle$ and $|1\rangle$. TR [24] F shows practical ways to perform additional state joins and splits, including an upgraded dependent type system to permit a few cases of splitting `EN` typed values.

Approximating Locus Scope. The type system approximates locus scopes. In rule T-IF, we use $\kappa \approx \kappa'$ as the approximate locus large enough to describe all possible qubits directly and indirectly mentioned in b and e . Such scope approximation might be over-approximated, which does not cause incorrectness in our proof system, while under-approximation is forbidden.

$$\begin{array}{c}
\text{P-FRAME} \\
\frac{\substack{\text{dom}(\sigma) \cap FV(\Omega, R) = \emptyset \\ FV(\Omega, e) \subseteq \text{dom}(\sigma) \\ \Omega; \sigma \vdash_g \{P\} e \{Q\}}}{\Omega; \sigma \uplus \sigma' \vdash_g \{P * R\} e \{Q * R\}}
\end{array}
\quad
\begin{array}{c}
\text{P-CON} \\
\frac{\substack{\sigma \preceq \sigma' \\ P \Rightarrow P' \\ \Omega; \sigma' \vdash_g \{P'\} e \{Q'\} \\ Q' \Rightarrow Q}}{\Omega; \sigma \vdash_g \{P\} e \{Q\}}
\end{array}$$

$$\begin{array}{c}
\text{P-OP} \\
\frac{\substack{\circ = op \vee o = \mu}}{\Omega; \{\kappa \bowtie \kappa' : \text{EN}\} \vdash_g \{\kappa \bowtie \kappa' \mapsto q\} \kappa \leftarrow \circ \{\kappa \bowtie \kappa' \mapsto \llbracket \circ \rrbracket^{|\kappa|} q\}}
\end{array}
\quad
\begin{array}{c}
\text{P-EXPC} \\
\frac{\substack{\Omega; \sigma \vdash_g \{P\} e[n/x] \{Q\}}}{\Omega; \sigma \vdash_g \{P\} \text{let } x = n \text{ in } e \{Q\}}
\end{array}$$

$$\begin{array}{c}
\text{P-MEA} \\
\frac{\substack{x \notin \text{dom}(\Omega) \\ \Omega[x \mapsto M]; \sigma \uplus \{\kappa : \text{EN}\} \vdash_c \{P[\mathcal{M}(x, n, \kappa)/y[0, n] \bowtie \kappa]\} e \{Q\}}}{\Omega; \sigma \uplus \{y[0, n] \bowtie \kappa : \text{EN}\} \vdash_c \{P\} \text{let } x = \text{measure}(y) \text{ in } e \{Q\}}
\end{array}$$

$$\begin{array}{c}
\text{P-IF} \\
\frac{\substack{FV(\Omega, b) = \kappa \\ \Omega; \{\kappa' : \text{EN}\} \vdash_M \{P[\mathcal{F}(b, \kappa, \kappa')/\kappa \bowtie \kappa']\} e \{Q\}}}{\Omega; \{\kappa \bowtie \kappa' : \text{EN}\} \vdash_g \{P\} \text{if } (b) e \{P[\mathcal{U}(\neg b, \kappa, \kappa \bowtie \kappa')/\kappa \bowtie \kappa'] * Q[\mathcal{U}(b, \kappa, \kappa \bowtie \kappa')/\kappa']\}}
\end{array}
\quad
\begin{array}{c}
\text{P-SEQ} \\
\frac{\substack{\Omega; \sigma \vdash_g \{P\} e_1 \{P'\} \\ \Omega; \sigma_1 \vdash_g \{P'\} e_2 \{Q\}}}{\Omega; \sigma \vdash_g \{P\} e_1 ; e_2 \{Q\}}
\end{array}$$

$$\begin{array}{c}
\text{P-LOOP} \\
\frac{n < n' \quad \Omega; \sigma \vdash_g \{P(j) \wedge j < n'\} \text{if } (b) e \{P(j+1)\}}{\Omega; \sigma[n/j] \vdash_g \{P(n)\} \text{for } j \in [n, n') \text{ && } b \{e\} \{P(n')\}}
\end{array}$$

 **Figure 11** Select proof rules.

For example, if we combine two Had-typed qubits in our system and transform the value to EN-type as $\frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle)$, this is an over-approximation since the two qubits are not entangled. Partially measuring the first qubit leaves the second qubit's value unchanged as $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$.

In addition to the above properties, we allow C-kind classical variables introduced by `let` to be evaluated in the type checking stage ⁹, while tracks M variables in Ω . Rule T-EXPC enforces that a classical variable x is replaced with its assigned value n in e , and classical expressions in e containing x are evaluated, so the proof system can avoid handling constants.

4.4 The Qafny Proof System

Every valid proof judgment $\Omega; \sigma \vdash_g \{P\} e \{Q\}$, shown in Figure 11, contains a pre- and post-condition predicates P and Q (syntax in Figure 7) for the statement e , satisfying the type restriction that $\Omega; \sigma \vdash_g e \triangleright \sigma'$; we also enforce that the predicate well-typed restrictions $\Omega; \sigma \vdash P$ and $\Omega; \sigma' \vdash Q$, meaning that all loci mentioned in P must be in the right forms and as elements in $\text{dom}(\sigma)$, introduced in TR [24] B. We state the restrictions as the typechecking constraint (TC) below:

$$TC(\sigma, P, Q) \triangleq \Omega; \sigma \vdash_g e \triangleright \sigma_1 \wedge \Omega; \sigma \vdash P \wedge \Omega; \sigma_1 \vdash Q$$

Rule P-CON describes the consequence rule where a well-formed pre- and post-conditions P and Q under σ is replaced by P' and Q' , well-formed under σ' . Under the new conditions, we enforce a new type constraint $TC(\sigma', P', Q')$. Rule P-FRAME is a specialized separation logic frame rule that separates the locus type environment and the quantum value to support local reasoning on quantum states. Rule M-FRAME in Figure 12 describes the predicate semantics

⁹ We consider all computation that only needs classical computers is done in the compilation time.

Predicate Model Rules:

M-MAP	$\Omega; \psi; \sigma; \varphi \uplus \{\kappa : \sum_j z_j c_j\rangle \langle \beta_j\}\models_g \kappa \mapsto \sum_j z_j c_j\rangle \langle \beta'_j\}$
M-LOCAL	$\Omega[x \mapsto M]; \psi[x \mapsto (r, v)]; \sigma; \varphi \models_g P \quad \text{if } \Omega; \psi; \sigma; \varphi \models_g P[(r, v)/x]$
M-FRAME	$\Omega; \psi; \sigma \uplus \sigma'; \varphi \uplus \varphi' \models_g P * Q \quad \text{if } \Omega; \psi; \sigma; \varphi \models_g P \text{ and } \Omega; \psi; \sigma'; \varphi' \models_g Q$

Transformation Rules:

M- \mathcal{F}	$\mathcal{F}(b, \kappa, \kappa') \mapsto q = \kappa' \mapsto \sum_j z_j \beta_j \langle c_j\rangle \beta'_j \rangle$
	where $\llbracket b \rrbracket^{ \kappa } q = q(\kappa, b) + q(\kappa, \neg b) \wedge q(\kappa, b) = \sum_j z_j c_j\rangle \beta_j \langle \beta'_j \rangle \wedge \forall j. c_j = \kappa $
M- \mathcal{U}	$\begin{aligned} \mathcal{U}(\neg b, \kappa, \kappa') &\mapsto \sum_j z_j c_j\rangle \eta_j + q(\kappa, \neg b) \\ * \mathcal{U}(b, \kappa, \kappa') &\mapsto \sum_j z'_j \beta_j \langle c_j\rangle \beta'_j \rangle \end{aligned} = \kappa' \mapsto \sum_j z'_j c_j\rangle \beta_j \langle \beta'_j \rangle + q(\kappa, \neg b) \quad \text{where } \forall j. \kappa = c_j $
M- \mathcal{M}	$\mathcal{M}(x, n, \kappa) \mapsto \sum_j z_j c\rangle \eta_j + q(\kappa, c \neq \kappa) = \kappa \mapsto \sum_j \frac{z_j}{\sqrt{r}} \eta_j * x = (r, \{c\}) \quad \text{where } n = c $

Figure 12 Predicate semantics.

of the separating conjunction $*$, where ψ is a local store mapping from M-kind variables to M-kind values (r, n) ; we require $\text{dom}(\psi) \subseteq \text{dom}(\Omega)$ and M-kind variables modeled by M-LOCAL. Besides predicate well-formedness, the predicate semantic judgment $(\Omega; \psi; \sigma; \varphi \models_g P)$ also ensures the states (φ) being well-formed $(\Omega; \sigma \vdash_g \varphi)$, defined as follows:

- **Definition 1** (Well-formed QAFNY state). A state φ is *well-formed*, written as $\Omega; \sigma \vdash_g \varphi$, iff $\text{dom}(\sigma) = \text{dom}(\varphi)$, $\Omega \vdash \text{dom}(\sigma)$ (all variables in φ are in Ω), and:
 - For every $\kappa \in \text{dom}(\sigma)$, s.t. $\sigma(\kappa) = \text{Nor}$, $\varphi(\kappa) = z |c\rangle \langle \beta|$ and $|\kappa| = |c|$ and $|z| \leq 1$; specifically, if $g = C$, $\beta = \emptyset$ and $|z| = 1$.¹⁰
 - For every $\kappa \in \text{dom}(\sigma)$, s.t. $\sigma(\kappa) = \text{Had}$, $\varphi(\kappa) = \frac{1}{\sqrt{2^n}} \otimes_{j=0}^{n-1} (|0\rangle + \alpha(r_j) |1\rangle)$ and $|\kappa| = n$.
 - For every $\kappa \in \text{dom}(\sigma)$, s.t. $\sigma(\kappa) = \text{EN}$, $\varphi(\kappa) = \sum_{j=0}^m z_j |c_j\rangle \langle \beta_j|$, and for all j , $|\kappa| = |c_j|$ and $\sum_{j=0}^m |z_j|^2 \leq 1$; specifically, if $g = C$, for all j , $\beta_j = \emptyset$ and $\sum_{j=0}^m |z_j|^2 = 1$.

Here, an M mode state, representing a computation living in an M mode context, has a relaxed well-formedness, where $\sum_{j=0}^m |z_j|^2 \leq 1$ and $\beta_j \neq \emptyset$. This is needed for describing the state inside the execution of a conditional body in rule S-IF in Section 4.2, where unmodified basis-kets are removed before the execution. There is a trick to utilizing the frozen stacks for promoting proof automation, as the modeling rule M-MAP equates two quantum values by discarding the frozen stack qubits, and we will see an example in Section 6.1.

The predicate syntax (Figure 7) introduces three locus predicate transformers \mathcal{F} , \mathcal{U} , and \mathcal{M} in the locus syntax category, but their semantics (Figure 12) essentially transform quantum states in the predicates, as we define them in equational style, explained below.

Assignment and Heap Mutation Operations. Rule P-EXPC describes C-kind variable substitutions. Rule P-OP is a classical separation logic style heap mutation rule for state preparations $\kappa \leftarrow op$ and oracles $\kappa \leftarrow \mu$, which analogize such operations as classical array map operations mentioned in Figure 2. Here, we discuss the cases when the state of the target loci $\kappa \uplus \kappa'$ is of type EN, while some other cases are introduced in TR [24] C.4. Each element in the array style pre-state q , for locus $\kappa \uplus \kappa'$, represents a basis-ket $z_j |c_j\rangle \eta_j$, with $|\kappa| = |c_j|$. Here, we first locate κ 's position basis $|c_j\rangle$ in each basis-ket of q , and then apply the operations op or μ to $|c_j\rangle$.

Quantum Conditionals. As in Section 3.4 and Figure 5, the key in designing a proof rule for a quantum conditional $\text{if } (b) \{e\}$ with its locus scope $\kappa \uplus \kappa'$, is to encode two transformers: \mathcal{F} and \mathcal{U} . In rule P-IF (Figure 11), we require the σ only contains locus $\kappa \uplus \kappa'$, which can be done

¹⁰ $|\kappa|$ and $|c|$ are the lengths of κ and c , and $|z|$ is the norm.

through P-FRAME. We then utilize $\mathcal{F}(b, \kappa, \kappa')$ to finish two tasks: (1) it computes b 's side-effects on the κ 's position bases ($\llbracket b \rrbracket^{|\kappa|} q$), and (2) it freezes all basis-kets that are irrelevant when reasoning about the body e . This freezing mechanism modeled by the equation M- \mathcal{F} (Figure 12) is accomplished at two levels: stashing all kets unsatisfying b ($q\langle \kappa, \neg b \rangle$) and moving κ 's position bases to basis stacks for the rest of basis-kets. After substituting $\kappa \bowtie \kappa'$ for $\mathcal{F}(b, \kappa, \kappa')$, besides expelling the parts not satisfying b , we also shrink the locus $\kappa \bowtie \kappa'$ to κ' , which in turn marked the κ 's position basis in every basis-ket inaccessible as κ is now invisible in the locus type environment.

After the body e 's proof steps, the post-state Q describes the computation result for κ' without the frozen parts. To reinstate the state for $\kappa \bowtie \kappa'$ by retrieving the frozen parts, we first substitute locus $\kappa \bowtie \kappa'$ for $\mathcal{U}(\neg b, \kappa, \kappa \bowtie \kappa')$ in P , which represents the unmodified part, unsatisfying b , in the pre-state. We also substitute κ' for $\mathcal{U}(b, \kappa, \kappa \bowtie \kappa')$ in Q , which represents the part satisfying b , evolved due to the execution of e . Rule M- \mathcal{U} (Figure 12) describes the predicate transformation, empowered by the locus construct \mathcal{U} , that utilizes the innate relation of separating conjunction and logical complement to assemble the previously unmodified and the evolved parts. Rule P-LOOP proves a **for** loop where $P(j)$ is the loop invariant parameterized over the loop counter j . Other rules are introduced in TR [24] C.4.

Measurement. A measurement (`let x = measure(y) in e`) collapses a qubit array y , binds an M kind outcome to x and restricts its usage in e . These statements usually appear in periodical patterns in many quantum algorithms, which users formalize as predicates to help verify algorithm properties. In rule P-MEA, we first select an n -length prefix bitstring c from one of range $y[0, n)$'s position bases; it then computes the probability r and assigns $(r, \{c\})$ to variable x . We then replace the locus $y[0, n) \bowtie \kappa$ in P with a locus predicate transformer $\mathcal{M}(x, n, \kappa)$ and update the type state Ω and σ by replacing $y[0, n) \bowtie \kappa$ with κ . The construct $\mathcal{M}(x, n, \kappa)$, with its transformation rule M- \mathcal{M} (Figure 12), is introduced to do exactly the two steps in Section 4.2 for describing measurement operations, i.e., we remove basis-kets not having c as $y[0, n)$'s position bases ($q\langle \kappa, c \neq \kappa \rangle$) and truncate $y[0, n)$'s position bases in the rest basis-kets.

$$\frac{\Omega[u \mapsto \text{M}]; \{x[0, n) : \text{EN}\} \vdash_c \{\mathcal{M}(u, n, x[0, n)) \mapsto C\} \{\} \{x[0, n) \mapsto D * E\}}{\Omega; \{y[0, n) \bowtie x[0, n) : \text{EN}\} \vdash_c \{y[0, n) \bowtie x[0, n) \mapsto C\} \text{ let } u = \text{measure}(y) \text{ in } \{\} \{x[0, n) \mapsto D * E\}}$$

$$C \triangleq \sum_{j=0}^{2^n-1} \frac{1}{\sqrt{2^n}} |a^j \% N\rangle |j\rangle \quad D \triangleq \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} |t + kp\rangle \quad E \triangleq p = \text{ord}(a, N) * u = (\frac{p}{2^n}, a^t \% N) * r = \text{rnd}(\frac{2^n}{p})$$

We show a proof fragment above for the partial measurement in Figure 6 line 8. The proof applies rule P-MEA by replacing locus $y[0, n) \bowtie x[0, n)$ with $\mathcal{M}(u, n, x[0, n))$. On the top, the pre- and post-conditions are equivalent, as explained below. In locus $y[0, n) \bowtie x[0, n)$'s state, for every basis-ket, range $y[0, n)$'s position basis is $|a^j \% N\rangle$; the value j is range $x[0, n)$'s position basis for the same basis-ket. Randomly picking a basis value $a^t \% N$ also filters a specific j in range $x[0, n)$, i.e., we collect any j having the relation $a^j \% N = a^t \% N$. Notice that modulo multiplication is a periodic function, which means that the relation can be rewritten $a^{t+kp} \% N = a^t \% N$, and p is the period order. Thus, the post-measurement state for range $x[0, n)$ can be rewritten as a summation of k : $\frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} |t + kp\rangle$. The probability of selecting $|a^j \% N\rangle$ is $\frac{r}{2^n}$. In the QAFNY implementation, we include additional axioms for these periodical theorems to grant this pre- and post-condition equivalence so that we can utilize QAFNY to verify Shor's algorithm.

4.5 Qafny Metatheory

We now present QAFNY’s type soundness and its proof system’s soundness and relative completeness. These results have all been verified in Coq. We prove our type system’s soundness with respect to the semantics, assuming well-formedness (TR [24] Definition 5 and Definition 1). The type soundness shows that our type system ensures the three properties in Section 4.3 and that the “in-place” style QAFNY semantics can describe all different quantum operations without losing generality because we can always use the equivalence rewrites to rewrite the locus state in ideal forms.

► **Theorem 2** (QAFNY type soundness). If $\Omega; \sigma \vdash_g e \triangleright \sigma'$ and $\Omega; \sigma \vdash_g \varphi$, then there exists φ' such that $(\varphi, e) \Downarrow \varphi'$ and $\Omega; \sigma' \vdash_g \varphi'$.

Our proof system is sound and relatively complete w.r.t. its semantics for well-typed QAFNY programs. Our system utilizes a subset of separation logic admitting completeness by excluding qubit array allocation and pointer aliasing. Since every quantum program in QAFNY converges, the soundness and completeness refer to the total correctness of the QAFNY proof system. $\psi(e)$ refers to that we substitute every variable $x \in \text{dom}(\psi)$ in e with $\psi(x)$.

► **Theorem 3** (proof system soundness). For any program e , such that $\Omega; \sigma \vdash_g e \triangleright \sigma'$ and $\Omega; \sigma \vdash_g \{P\} e \{Q\}$, and for every ψ and φ , such that $\Omega; \sigma \vdash_g \varphi$ and $\Omega; \psi; \sigma; \varphi \models_g P$, there exists a state φ' , such that $(\varphi, \psi(e)) \Downarrow \varphi'$ and $\Omega; \sigma' \vdash_g \varphi'$ and $\Omega; \psi; \sigma'; \varphi' \models_g Q$.

► **Theorem 4** (proof system relative completeness). For a well-typed program e , such that $\Omega; \sigma \vdash_g e \triangleright \sigma'$, $(\varphi, e) \Downarrow \varphi'$ and $\Omega; \sigma \vdash_g \varphi$, and for all predicates P and Q such that $\Omega; \emptyset; \sigma; \varphi \models_g P$ and $\Omega; \emptyset; \sigma'; \varphi' \models_g Q$, we have $\Omega; \sigma \vdash_g \{P\} e \{Q\}$.

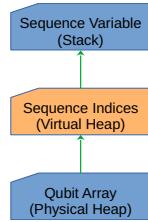
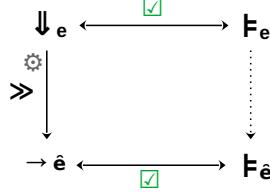
5 Qafny Compilation and Implementation Evaluation

Here, we focus on the QAFNY proof system compilation to a subset of separation logic.

5.1 Translation from Qafny to Separation Logic

The QAFNY types and loci are extra annotations associated with QAFNY programs and predicates for proof automation. In the QAFNY implementation in Dafny, these annotations are not present – qubits are arranged as simple array structures without extra metadata such as locus types. This section shows how QAFNY annotations can be safely erased with no loss of expressiveness, e.g., loci are represented as virtual-level dynamic sequences without types, and equational rewrites are compiled with extra operations in the compiled language. We present a compilation algorithm that converts from QAFNY to SEP, a C-like language admitting a subset of an array separation logic proof system.

Target Compilation Language. SEP is based on a variant of the separation logic introduced by Yang and O’Hearn [55], which is sound and complete. Mainly, we utilize the allocation (`alloc`), heap lookup and mutation operations (`mutate`) in the work, with three additional operations (marked red in Figure 13). In SEP, program states are divided into virtual and physical levels in Figure 14. Every SEP program starts with a physical qubit array, analogous to physical heap structures. The program operations are applied to qubit sequences of array indices (A), representing a collection of physical qubit locations that live at the virtual level. SEP permits immutable program variables (x) representing these sequences.

$$\begin{aligned}
 A, B &::= \bar{n} \\
 \tilde{o}p &::= \mu \mid op \\
 \tilde{e} &::= x = \text{alloc}(A) \\
 &\quad \mid \text{mutate}(n, \tilde{o}p, x) \\
 &\quad \mid (r, n) = \text{pick}(x, m) \\
 &\quad \mid \text{filter}(x, b) \\
 &\quad \mid \text{amp}(x, r) \\
 &\quad \dots
 \end{aligned}$$
Figure 13 SEP Syntax.**Figure 14** Heap Layout.**Figure 15** Compilation Proof Diagram.

An allocation $x = \text{alloc}(A)$ allocates a new array x that copies A 's content and has the same length as A , while a heap mutation $\text{mutate}(n, \tilde{o}p, x)$ mutates the first n elements of the array pointed to by x , by applying the operation $\tilde{o}p$. Operations pick , filter , and amp are variations of heap lookups and mutations. $(r, n) = \text{pick}(x, m)$ measures the first m qubits in x , with the outcome $n = \{\text{c}\}$ and its probability r , similar to the computation in S-MEA (Figure 8). $\text{filter}(x, b)$ mutates x 's pointed-to quantum value by filtering out basis-kets that are not satisfying b , and $\text{amp}(x, r)$ multiples r to every basis-ket in x 's quantum value. Yang and O'Hearn maintain completeness by carefully designing the proof rules so heap mutations do not modify pointer references. SEP ensures the same property by immutable variables, i.e., if a sequence changes, we allocate a new array and a new variable pointing to the array. For example, to join two loci represented by two sequences A and B , respectively pointed to by variables x and y , we allocate a new space for the two sequences' concatenation ($A@B$); so we compile the join to $u = \text{alloc}(A@B)$. We must abandon using x and y after the join and only refer to u in the following computation.

Compilation Procedure. As shown in Figure 15, we compile the QAFNY language to SEP and achieve the proof system compilation through the proof system completeness in QAFNY and SEP. For every QAFNY program, we translate the program and states to SEP. Then, every provable triple in QAFNY can be translated to a provable SEP triple through the language translation from QAFNY to SEP as the diagram (Figure 15). The compilation is defined by extending QAFNY's typing judgment thusly: $\Omega; \sigma \vdash_g (\theta, \varphi, e) \gg (\theta, \tilde{\psi}, \tilde{\varphi}, \tilde{e})$. We include an initial QAFNY state φ , the output local store $(\tilde{\psi})$, mapping variables to qubit location sequences, and state $(\tilde{\varphi})$, mapping from locations to quantum values. θ and θ' are maps from locus locations in φ to SEP qubit locations in $\tilde{\varphi}$.

Here, we explain the rules for compilation by examples of compiling the QAFNY operations to SEP. The locus rewrites (Section 4.3) are compiled to the array allocations, such as the join operation above. Additionally, the split of a locus is compiled to two consecutive allocations of two sequences, respectively representing the two split result loci. In compiling a measurement statement (`let x=measure(y) in ...`), where y locates in the locus $y[0, n] \sqcup \kappa$, let's assume that the locus is mapped in θ by sequence $[0, n+m]$, pointed to by u in $\tilde{\psi}$, while the range $y[0, n]$ is mapped by the sequence $[0, n]$; the operation is compiled to:

$$(r, p) = \text{pick}(u, m) ; \text{filter}(u, u[0, n] = p) ; \text{amp}(u, \sum_j \frac{z_j}{\sqrt{r}}) ; t = \text{alloc}([n, n+m])$$

We first pick a key p , filter out the basis-kets whose $u[0, n]$'s position bases are not p , normalize the amplitudes of the remaining basis-kets (Section 4.4), and allocate a new space t for the quantum residue after the measurement. We also update κ in θ to map to the newly allocated space of t instead of $[n, n+m]$. We compile an operation $x[0, n] \sqcup y[0] \leftarrow (x < 5) @ y[0]$ with its initial state φ ($C = \frac{1}{\sqrt{2^n}} \otimes_{j=0}^{n-1} (|0\rangle + |1\rangle)$) to SEP, with $D = \sum_{j=0}^{2^n-1} |j\rangle |j < 5\rangle$. Such an operation computes the Boolean comparison of $x < 5$ and stores the value to $y[0]$.

Algorithm	Qafny		QBricks		SQIR	
	Runtime (sec)	LOC	Runtime (sec)	LOC	Runtime (sec)	LOC
GHZ	14.2	16	-	-	141	119
Deutsch-Jozsa	8.3	13	74	108	163	408
Grover's search	26.7	27	253	233	148	1018
Shor's algorithm	36.3	36	1328	1163	1244	8464

Figure 16 Running time (include theory loading) & LOC comparison, in an i7 Ubuntu Mach. 8G RAM; -: no data.

Algorithm	Runtime (sec)	LOC
Controlled GHZ	6.4	12
Quantum Walk	43.1	49

Figure 17 QAFNY data for case studies in Section 6.

$$\begin{aligned} \varphi &= \{x[0, n] : C, y[0] : |0\rangle\} \\ x[0, n] \sqcup y[0] &\leftarrow (x < 5) @ y[0] \quad \Rightarrow \quad \tilde{\psi} = \{p : [0, n], t : \{n\}\}, \tilde{\varphi} = \{[0, n] : C * \{n\} : |0\rangle\} \\ \varphi' &= \{x[0, n] \sqcup y[0] : D\} \end{aligned}$$

$$u = \text{alloc}([0, n+1)); \text{mutate}(n+1, u[0, n] < 5 @ u[n], u)$$

$$\tilde{\psi}' = \tilde{\psi} \cup \{u : [n+1, 2n+2]\}, \tilde{\varphi}' = \tilde{\varphi} \cup \{[n+1, 2n+2] : D\}$$

After the compilation, we create two local variables p and t to represent the loci x and y , mapping to sequences $[0, n]$ and $\{n\}$. We then add $u = \text{alloc}([0, n+1])$ allocating a new space $[n+1, 2n+2]$ to join the two loci. The post-state contains a new variable u , pointing to the concatenated new sequence $[n+1, 2n+2]$. The old arrays p and t are still in the stores, but we refer to the locus $x[0, n] \sqcup y[0]$ as the newly allocated space in the following computation by mapping the locus to the new space in θ' . As a future work, we will prove the proof system compilation correctness from QAFNY to SEP, proof strategy in Figure 15.

5.2 Implementation and Comparison to Existing Quantum Verifications

We have implemented a prototype QAFNY to Dafny compiler, which faithfully respects the presented QAFNY to SEP compilation algorithm. To validate the soundness of the compiler implementation, we create many test cases for the compiler. We then insert a number of bugs in these test cases; Qafny has been able to detect all of them. Dafny's proof engine cannot be used to verify arbitrary separation-logic assertions because it only has an implicit frame rule implementation that allows users to set up variables that can be modified in a function. However, QAFNY only requires a subset of separation logic, and the QAFNY loci disjoint property and non-aliasing guarantee ensures that the QAFNY separation conjunctions are captured by Dafny's implicit frames when we compile QAFNY to Dafny. We utilize the QAFNY to Dafny compiler as an automated verification framework to verify six quantum programs, shown in Section 6 and Figure 17.

There were two main approaches to verifying quantum programs: program and measurement-based. The former views quantum program transitions as a state machine and verifies the inductive relations among transitions, while the latter focuses on the relations between quantum program measurements and the post-processing classical components – they typically view quantum components as black boxes with specifications.

QAFNY is program-based and other program-based mechanized frameworks for formally verifying quantum programs, including Qwire [41], SQIR [16, 15] (an upgrade of Qwire), and QBricks [3], provide libraries in an interactive proof assistant to guide users for building inductive proofs for quantum programs; each has verified 7-10 quantum programs. The core of SQIR and QBricks, as well as other executable quantum verification platforms, is a circuit-description language. Verifying a program in these frameworks inductively builds a unitary or density matrix as the program's quantum circuit semantic interpretation. For example, to verify Shor's algorithm, both QBricks and SQIR require inductive proofs based on elementary circuit gates to derive the unitary or density matrix semantics of different sub-components, including state preparations, oracles, and QFT^{-1} gates. Additionally, program

verification in these frameworks requires the development of theories and tactics to capture program properties, which usually involves the proof of additional theorems. This approach is qualitatively different from QAFNY, where program verification involves embedding assertions in a program for completing a proof. QAFNY identifies a few program structures, such as oracles and quantum conditionals, and formulates inductive patterns involving these quantum components as proof rules. These rules interact with quantum program operations and states for deriving verification proofs, so proofs are largely automated based on this small set of structures.

Figure 16 shows a quantitative comparison of QAFNY, regarding theory/proof statement running time and numbers of lines (LOC), with respect to QBricks and SQIR for verifying several quantum programs. The results show that QAFNY has the shortest running time and LOCs for verifying programs with our automated proof engine. The QBricks verification has better LOCs but a similar running time compared to SQIR. SQIR provides a complete verification [38] by proving every mathematical theorem involved in the verification, so its verification proofs are longer than QBricks; QBricks performs better by providing some automatic tactics for sequence operations ($e ; e$) and taking many math theorems as assumptions without proof. In testing the two frameworks, we found that the previous claim [16, 15] that rigorous quantum proofs are one-time cost is problematic because inductive theorem provers update constantly. Once an update happens, users might need to fix the proofs (not programs or specifications) in their history code, e.g., our researcher spent three days fixing minor bugs in the proofs in SQIR and QBricks due to Coq and Why3 version issues. Moreover, a new program verification in SQIR typically required detailed proofs of additional theorems beyond the program specifications.

QAFNY provides fast prototyping, where we apply the automated verification mechanisms in many classical systems [40, 22] to verify quantum programs and save programmers' effort. Verifying programs in many inductive theorem provers takes weeks and months to finish, while the same tasks in QAFNY cost researchers a few days due to the QAFNY features mentioned above. The fast prototyping in QAFNY can also help users to explore and understand new quantum program patterns such as the two case studies in Section 6, whose running time and LOCs are shown in Figure 17. Compared to the data of well-known algorithms in Figure 16, the data for verifying the new programs do not show a significant difference, showing QAFNY's ability to explore new algorithm behaviors without proving many new theories, which usually appears in the above quantum verification frameworks.

6 Case Studies

With two examples, we show QAFNY as a rapid prototyping tool for quantum programs.

6.1 Controlled GHZ: Composing Quantum Algorithms from Others

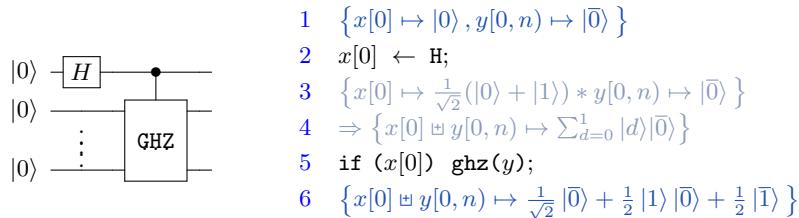


Figure 18 Controlled GHZ circuit and proof. $\text{ghz}(y)$ is in Fig. 3. Lines 3-4 automatically inferred.

Automated verification frameworks such as Dafny encourage programmers to build program proofs based on the reuse of subprogram proofs. However, this perspective is more or less overlooked in previous quantum proof systems. In SQIR, for example, verifying the correctness of a *controlled GHZ*, a simple circuit constructed by extending GHZ with an extra control qubit, requires generalizing the GHZ circuit to any arbitrary inputs. In QAFNY, users do not need to do this, as shown here.

Figure 18 provides a proof of the Controlled GHZ algorithm based on a proven GHZ method in Figure 3c. The focal point is the quantum conditional on line 5. For verifying a GHZ circuit, its input is an n -qubit `Nor`-typed value of all $|0\rangle$, but the given value, in line 4, is an `EN`-typed entanglement $\sum_{d=0}^1 |d\rangle|\bar{0}\rangle$. Here is where SQIR gets stuck. In QAFNY, we automatically verify the proof by rule P-IF and the equivalence relation to rewrite a singleton EN value to a `Nor` one, as $\sum_{j=0}^0 z_j|c_j\rangle \equiv z_0|c_0\rangle$. The detailed proof for the conditional is given below, where $U(b) = \mathcal{U}(b, x[0], \kappa)$ and $\kappa = x[0] \sqcup y[0, n]$.

$$\begin{array}{c}
 \Omega; \{y[0, n] : \text{Nor}\} \vdash_m \left\{ y[0, n] \mapsto |\bar{0}\rangle|\bar{1}\rangle \right\} \text{ghz}(y) \left\{ y[0, n] \mapsto \sum_{d=0}^1 \frac{1}{\sqrt{2}} |\bar{d}\rangle|\bar{1}\rangle \right\} \\
 \hline
 \Omega; \{y[0, n] : \text{EN}\} \vdash_m \left\{ \mathcal{F}(x[0], y[0, n]) \mapsto \sum_{d=0}^1 |d\rangle|\bar{0}\rangle \right\} \text{ghz}(y) \left\{ y[0, n] \mapsto \sum_{d=0}^1 \frac{1}{\sqrt{2}} |\bar{d}\rangle|\bar{1}\rangle \right\} \quad \text{EQ} \\
 \hline
 \Omega; \{\kappa : \text{EN}\} \vdash_c \left\{ \kappa \mapsto \sum_{d=0}^1 |d\rangle|\bar{0}\rangle \right\} \text{if } (x[0]) \text{ ghz}(y) \left\{ \mathcal{U}(\neg x[0]) \mapsto \sum_{d=0}^1 |d\rangle|\bar{0}\rangle * U(x[0]) \mapsto \sum_{d=0}^1 \frac{1}{\sqrt{2}} |\bar{d}\rangle|\bar{1}\rangle \right\} \quad \text{P-IF} \\
 \hline
 \Omega; \{\kappa : \text{EN}\} \vdash_c \left\{ \kappa \mapsto \sum_{d=0}^1 |d\rangle|\bar{0}\rangle \right\} \text{if } (x[0]) \text{ ghz}(y) \left\{ \kappa \mapsto \frac{1}{\sqrt{2}} |\bar{0}\rangle + \frac{1}{2} |1\rangle|\bar{0}\rangle + \frac{1}{2} |\bar{1}\rangle \right\}
 \end{array}$$

After rule P-IF is applied, locus $y[0, n]$'s value is rewritten to a `Nor` type value on the top as $|\bar{0}\rangle|\bar{1}\rangle$, where $|\bar{0}\rangle$ is n qubits and $|\bar{1}\rangle$ is frozen in the stack. Since two values are equivalent as QAFNY discards stacks, $|\bar{0}\rangle|\bar{1}\rangle$ is equivalent to $|\bar{0}\rangle$, which satisfies the input condition for GHZ, so all proof obligations introduced to invoke the `ghz` method are discharged.

6.2 Case Study: Understanding Quantum Walk

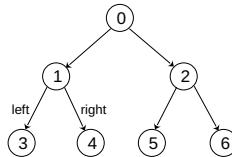


Figure 19 Tree Structure.

Quantum walk [54, 4, 53] is a quantum version of the classical random walk [36] and an important framework for developing quantum algorithms. However, most quantum walk analyses are based on Hamiltonian simulation, which deters many computer programmers from the quantum walk framework. Here, we show that the discrete-time quantum walk, at its very least, is a quantum version of breadth-first search; thus, many algorithms [4] based on Quantum walk can be understood as performing search algorithms in the quantum setting.

Figure 20 lists the proof outline for the core loop of a discrete-time quantum walk algorithm to traverse a complete binary tree (structure in Figure 19); each node has a unique key. The m -depth nodes in the tree have keys $j \in [2^m - 1, 2^{m+1} - 2]$, which form a sequence

```

 $q(j) \triangleq \sum_{i=0}^{2^{(j+1)}-3} z_i \langle \lfloor \log(i+1) \rfloor | \text{pat}(i, j) \rangle |i+1\rangle |d_i\rangle \text{ where } \text{pat}(i, j) \triangleq |0\rangle^{\otimes \lfloor \log(i+1) \rfloor} |1\rangle^{\otimes (j - \lfloor \log(i+1) \rfloor)}$ 
1  $\{x[0, t] \mapsto \frac{1}{\sqrt{2^t}} \otimes_{i=0}^{t-1} (|0\rangle + |1\rangle) * y[0, m] \mapsto |\bar{0}\rangle * h[0, n] \mapsto |\bar{0}\rangle * u[0] \mapsto |0\rangle * m = 2^t \cdot m < n\}$ 
2  $\Rightarrow \{x[0, t] \bowtie y[0, 0] \bowtie h[0, n] \bowtie u[0] \mapsto \sum_{k=0}^{2^t-1} \frac{1}{\sqrt{2^t}} |k\rangle |\bar{0}\rangle |0\rangle * y[0, m] \mapsto |\bar{0}\rangle * m = 2^t \cdot m < n\}$ 
3 for  $j \in [0, m)$  &&  $(x[0, t] < j+1) @ y[j]$ 
4  $\{x[0, t] \bowtie y[0, j] \bowtie h[0, n] \bowtie u[0] \mapsto q(j) + \sum_{k=j}^{2^t-1} \frac{1}{\sqrt{2^t}} |k\rangle |\bar{0}\rangle |\bar{0}\rangle |0\rangle * y[j, m] \mapsto |\bar{0}\rangle\}$ 
5  $\{u[0] \leftarrow \text{H};$ 
6  $\text{if } (u[0]) \text{ left}(\lfloor \log(j+1) \rfloor, h[0, n]);$ 
7  $\text{if } (\neg u[0]) \text{ right}(\lfloor \log(j+1) \rfloor, h[0, n));\}$ 
8  $\{x[0, t] \bowtie y[0, m] \bowtie h[0, n] \bowtie u[0] \mapsto q(m)\}$ 

```

Figure 20 Quantum walk reachable node verification for a complete binary tree. **left** and **right** reach corresponding children. $q(j)$ is a quantum value with var j . $i+1$ is a node key in a tree.

from left to right in depth m -th, such as the sequence 3, 4, 5, 6 in depth 2 in Figure 19. Thus, a node (key j) has a depth $m = \lfloor \log(j+1) \rfloor$, and its **left** and **right** children have keys $2 \cdot j + 1$ and $2 \cdot j + 2$, respectively representing the **left** and **right** operation semantics in Figure 20, i.e., for any basis $|j\rangle$, with m being the depth and j a node key, the outputs of applying **left** and **right** are $|2 \cdot j + 1\rangle$ and $|2 \cdot j + 2\rangle$, respectively¹¹.

The algorithm in Figure 20 requires four quantum ranges: a t -qubit range x in superposition, an m -qubit range where $y[j]$'s position bases keep the result of evaluating $x[0, t] < j+1$ for j -th loop step, an n -qubit node register h storing the node keys, and a single qubit u acting as the random walk coin and determining the moving direction of the next step (1 for the left and 0 for the right). In line 2, we merge the ranges x , u , and h as the locus $x[0, t] \bowtie y[0, 0] \bowtie h[0, n] \bowtie u[0]$ ($y[0, 0]$ is empty); at each loop step (lines 3-7), we entangle a qubit in the range y into the locus. Finally, at line 8, the loop program entangles all these ranges together as a locus $x[0, t] \bowtie y[0, m] \bowtie h[0, n] \bowtie u[0]$.

In the j -th loop step, we abbreviate locus $x[0, t] \bowtie y[0, j] \bowtie h[0, n] \bowtie u[0]$ as $\kappa(j)$, and locus $\kappa(j)$'s state value is split into two basis-ket sets, separated by $+$ in Figure 20 line 4. To verify a step, we first split the $y[j]$ qubit, having position basis $|0\rangle$, from range $y[j, m]$, and merge the qubit into $\kappa(j)$. The split rewrites are given as:

$$\begin{aligned}
& \{ \kappa(j) \mapsto \sum_{i=0}^{2^{(j+1)}-3} z_i \langle \lfloor \log(i+1) \rfloor | \text{pat}(i, j) \rangle |i+1\rangle |d_i\rangle + \sum_{k=j}^{2^t-1} \frac{1}{\sqrt{2^t}} |k\rangle |\bar{0}\rangle |\bar{0}\rangle |0\rangle * y[j, m] \mapsto |\bar{0}\rangle \} \\
& \equiv \{ \kappa(j+1) \mapsto \sum_{i=0}^{2^{(j+1)}-3} z_i \langle \lfloor \log(i+1) \rfloor | \text{pat}(i, j) \rangle |0\rangle |i+1\rangle |d_i\rangle + \sum_{k=j}^{2^t-1} \frac{1}{\sqrt{2^t}} |k\rangle |\bar{0}\rangle |0\rangle |\bar{0}\rangle |0\rangle * y[j+1, m] \mapsto |\bar{0}\rangle \} \\
& \equiv \{ \kappa(j+1) \mapsto q'(|0\rangle) + \sum_{k=j}^{2^t-1} \frac{1}{\sqrt{2^t}} \delta(k, 0) * y[j+1, m] \mapsto |\bar{0}\rangle \}
\end{aligned}$$

At the last rewrite above, we abbreviate the first part of $\kappa(j+1)$'s value to be $q'(|0\rangle)$, and the second part to be $\sum_{k=j}^{2^t-1} \frac{1}{\sqrt{2^t}} \delta(k, 0)$ where $\delta(k, c) = |k\rangle |\bar{0}\rangle |c\rangle |\bar{0}\rangle |0\rangle$. Below, we show the proof steps (only the pre-condition transitions) for a conditional step in Figure 20 line 3, which can be divided into two small steps. Here, e is the conditional body in lines 5-7, and we apply P-FRAME to frame out locus $y[j+1, m]$ from the states, so the bottom state only refers to the locus $\kappa(j+1)$. We further split the second part above ($\sum_{k=j}^{2^t-1} \frac{1}{\sqrt{2^t}} \delta(k, 0)$) into two basis-ket sets: $\frac{1}{\sqrt{2^t}} \delta(j, 0)$ and $\sum_{k=j+1}^{2^t-1} \frac{1}{\sqrt{2^t}} \delta(k, 0)$.

¹¹The tree structure is a simplification; comprehensive implementations use Szegedy walk encoding [30].

$$\frac{\Omega; \{h[0, n] \sqsubseteq u[0] : \text{EN}\} \vdash_{\mathbb{M}} \left\{ h[0, n] \sqsubseteq u[0] \mapsto \sum_{i=0}^{2^{(j+1)-3}} z_i |i+1\rangle |d_i\rangle \widehat{\beta} + \frac{1}{\sqrt{2^t}} |\bar{0}\rangle |0\rangle \widehat{\beta}' \right\} e \{ \dots \}}{\Omega; \{\kappa\langle j+1 \rangle : \text{EN}\} \vdash_{\mathbb{M}} \left\{ \kappa\langle j+1 \rangle \mapsto q'(|1\rangle) + \frac{1}{\sqrt{2^t}} \delta\langle j, 1 \rangle + \sum_{k=j+1}^{2^t-1} \frac{1}{\sqrt{2^t}} \delta\langle k, 0 \rangle \right\} \text{ if } ((x[0, t) < j+1) @ y[j]) e \{ \dots \}}$$

$$\frac{\Omega; \{\kappa\langle j+1 \rangle : \text{EN}\} \vdash_{\mathbb{M}} \left\{ \kappa\langle j+1 \rangle \mapsto q'(|0\rangle) + \frac{1}{\sqrt{2^t}} \delta\langle j, 0 \rangle + \sum_{k=j+1}^{2^t-1} \frac{1}{\sqrt{2^t}} \delta\langle k, 0 \rangle \right\} \text{ if } ((x[0, t) < j+1) @ y[j]) e \{ \dots \}}{\Omega; \{u[0] \sqsubseteq h[0, n] : \text{EN}\} \vdash_{\mathbb{M}} \left\{ u[0] \sqsubseteq h[0, n] \mapsto \sum_{i=0}^{2^{(j+1)-3}} \frac{1}{\sqrt{2}} z_i |d_i\rangle |i+1\rangle \widehat{\beta} + \sum_{i=0}^{2^{(j+1)-3}} \frac{1}{\sqrt{2}} z_i |d_i+1\rangle |i+1\rangle \widehat{\beta} + \frac{1}{\sqrt{2^{t+1}}} |\bar{0}\rangle |\bar{0}\rangle \widehat{\beta}' + \frac{1}{\sqrt{2^{t+1}}} |1\rangle |\bar{0}\rangle \widehat{\beta}' \right\}}$$

We split the P-If proof step (Line 3 in Figure 20) into two small steps above. The bottom step represents the first half of the \mathcal{F} transformer application (Figure 11) in P-If. It views the Boolean guard $(x[0, t) < j+1) @ y[j]$ as an oracle application and for every basis-ket in the locus $\kappa\langle j+1 \rangle$, we compute the Boolean value $x[0, t) < j+1$ and store it to $y[j]$'s position bases. Unlike the simple Boolean guards appearing in Figures 3c and 6, the Boolean guard here has side-effects that modify $y[j]$'s position bases. $\kappa\langle j+1 \rangle$'s value is split into three basis-ket sets separated by $+$. In the set $q(|0\rangle)$, range $x[0, t)$'s position bases have the form $|\lfloor \log(i+1) \rfloor\rangle$ (the depth of a node key $i+1$) and the bases' natural number interpretations are smaller than $j+1$; in the sets $\frac{1}{\sqrt{2^t}} \delta\langle j, 0 \rangle$ and $\sum_{k=j+1}^{2^t-1} \frac{1}{\sqrt{2^t}} \delta\langle k, 0 \rangle$, range $x[0, t)$'s position bases are $|j\rangle$ and $|k\rangle$ ($j < k$). The former's natural number interpretation is less than $j+1$, while the latter is not. Thus, we flip $y[j]$'s position bases of the two sets after applying the bottom rule above while leaving the third set unchanged.

The middle step in the above P-If proof step rules out the basis-ket set $\sum_{k=j+1}^{2^t-1} \frac{1}{\sqrt{2^t}} \delta\langle k, 0 \rangle$, because the $y[j]$'s position bases are all 0; then, we push bases $|\lfloor \log(i+1) \rfloor\rangle |\text{pat}(i, j)\rangle |0\rangle$ and $|k\rangle |\bar{0}\rangle |0\rangle$ to the frozen stacks as $\widehat{\beta}$ and $\widehat{\beta}'$, respectively for the remaining two sets.

$$\frac{\Omega; \{u[0] \sqsubseteq h[0, n] : \text{EN}\} \vdash_{\mathbb{M}} \left\{ u[0] \sqsubseteq h[0, n] \mapsto \sum_{i=0}^{2^{(j+1)-3}} z_i |d_i\rangle |i+1\rangle \widehat{\beta} + \frac{1}{\sqrt{2^t}} |\bar{0}\rangle |\bar{0}\rangle \widehat{\beta}' \right\} u[0] \leftarrow \text{H} \quad \left\{ u[0] \sqsubseteq h[0, n] \mapsto \sum_{i=0}^{2^{(j+1)-3}} \frac{1}{\sqrt{2}} z_i |d_i\rangle |i+1\rangle \widehat{\beta} + \sum_{i=0}^{2^{(j+1)-3}} \frac{1}{\sqrt{2}} z_i |d_i+1\rangle |i+1\rangle \widehat{\beta} + \frac{1}{\sqrt{2^{t+1}}} |\bar{0}\rangle |\bar{0}\rangle \widehat{\beta}' + \frac{1}{\sqrt{2^{t+1}}} |1\rangle |\bar{0}\rangle \widehat{\beta}' \right\}}$$

For the H application in line 5 (Figure 20), we first rewrite the locus, in the pre- and post-states, from $h[0, n] \sqsubseteq u[0]$ to $u[0] \sqsubseteq h[0, n]$; shown as the proof triple above. There is a hidden uniqueness assumption ¹² for all basis-kets in $q(j)$ (Figure 20): $\forall z\beta |d_i\rangle \in q(j) \Rightarrow \forall z' . z'\beta |d_i+1\rangle \notin q(j)$, i.e., if we truncate the $u[0]$ qubit, every basis is still unique in $q(j)$. For each basis-ket, the H application duplicates the non- $u[0]$ part, with the flip of $u[0]$'s position bases. During the process, the amplitude of each basis-ket is reduced by $\frac{1}{\sqrt{2}}$. The **left** and **right** in lines 6 and 7 then move the node key (range $h[0, n]$'s position basis) of each basis-ket to its **left** and **right** child, depending on the coin bit stored as $u[0]$'s position basis; thus, the uniqueness property is preserved (left and right children always have different keys). Remember that the number of basis-kets is doubled in the H gate application; after the j -th loop step, all $(j-1)$ -th depth nodes become j -th depth and the two root node basis-kets ($\frac{1}{\sqrt{2^{t+1}}} |0\rangle |\bar{0}\rangle \widehat{\beta}'$ and $\frac{1}{\sqrt{2^{t+1}}} |1\rangle |\bar{0}\rangle \widehat{\beta}'$) become 1-st depth nodes; thus, the state, after j -th loop step, is in superposition containing all nodes up to j -th depth, except the root node.

The above applications also show the necessity of frozen basis stacks. When applying the conditional, we hide $x[0, t)$'s position bases to frozen basis stacks, and there are $2^{(j+1)-3}$ different such stacks. We need to record the position bases in the frozen basis stacks; so, 1) when we apply the H gate, we know which basis-kets are associated with a specific position basis; 2) once the conditional is over, the position bases can be retrieved.

The verification in Figure 20 describes the basic property of the quantum walk algorithm framework. The biggest advantage of the framework is to permit the manipulation of different quantum applications on different tree nodes in each loop step, which is why many algorithms [5, 4, 28, 1] have been developed based on it.

¹²In the Dafny implementation, this needs to be an explicit assumption given by the users.

7 Related Work

This section gives related work beyond the discussion in Section 5.2.

Measurement-based Quantum Proof Systems. Except for the works in Section 5.2, previous quantum proof systems are measurement-based, including quantum Hoare logic [56, 26, 10, 57], quantum separation logic (QSL) [20, 59], quantum relational logic [25, 52], and probabilistic Hoare logic for quantum programs [19], informed the QAFNY development. They differ from QAFNY in three main respects, however: 1) their conditionals are solely classical, while QAFNY has quantum conditionals; 2) they mainly focus on the probabilistic relations between the quantum measurement results and classical components and view quantum program components as black boxes specified by Hilbert spaces or density matrices; and 3) most of them have no mechanized implementations, and they do not have a quantum program compiler. The verification procedure in these frameworks, to some extent, shows the possibility of verifying mainly hybrid classical-quantum (HCQ) programs by requiring the input of black-boxed and verified quantum program components.

QSL [20] develops a new separation logic theory (with no executable proof examples, however) for Hilbert spaces and classical controls, mainly for verifying HCQ programs by black-boxing quantum components. This differs from the QAFNY system, based on classical separation logic for classical array operations. QSL is based on a notion of frame rules that split a tensor product state into two parts, similar to our `Had` typed state split equation. However, they do not specify when and how a quantum state separation may happen. As in Section 3, in many cases, quantum state separation is not trivial and might not be automatically inferred by a proof engine.

Liu et al. [26] contains an example verification for Grover’s search algorithm based on the SQIR inductive verification style, albeit with worse proof automation (3184 LOC vs. 1018 LOC in Figure 16). CoqQ [60] provides a mechanized automated verification framework for HCQ programs. However, their proof automation is to connect quantum and classical components, i.e., they view quantum circuit components as black-boxes. By giving pre- and post-conditions, they perform proof automation on verifying HCQ programs that view the quantum components as sub-procedures. There are some examples in CoqQ to verify quantum components, but they are handled in the same style as SQIR and QBricks above. See TR [24] D.

Classical Proof Systems. We are informed by separation logic, as articulated in the classic paper [42], and other papers as well [18, 50, 34, 58, 27, 46]. Primarily, we show a compilation from QAFNY to SEP, representing a subset of separation logic admitting sound and completeness [55], which was also studied by [51, 9]. The QAFNY implementation is compiled to Dafny [21], a language designed to simplify writing correct code. The natural proof methodology [27, 37, 29] informs the QAFNY development, where it embeds the proofs of data-structures to a recursive search problem.

8 Conclusion and Future Work

We present QAFNY, a system for expressing and automatically verifying quantum programs that can be compiled into quantum circuits. We develop a proof system that views quantum operations as classical array aggregate operations, e.g., viewing quantum measurements as array filters, so that we can map the proof system, which is sound and complete with respect

to the QAFNY semantics for well-typed programs, to classical verification infrastructure. We implement a prototype compiler in Dafny and use it to verify several quantum programs. We believe that programmers can utilize QAFNY to develop quantum algorithms and verify them through our automated verification engine with a significant saving of human effort, as demonstrated in Section 6. The QAFNY language is universal in terms of the power of expressing quantum programs since all gates in the universal RzQ gate set {H, X, RZ, CNOT} [33] are definable. However, being able to define all possible quantum programs does not mean that we can utilize QAFNY to verify all quantum programs, especially HCQ programs, automatically. Verifying HCQ programs requires the full power of quantum mixed states, i.e., users might want to know the probabilistic output of executing a quantum program with a quantum input state being associated with a probability. Verifying all such programs requires a powerful classical probability distribution library beyond the current scope of QAFNY, although the existing QAFNY implementation does include a restricted library for reasoning about probability distributions [32] that can verify some HCQ programs.

In future work, we plan to build and verify a complete QAFNY implementation in Dafny; especially, we intend to enhance the probability distribution libraries to automatically verify more HCQ programs. We also want to show the soundness proof of the implementation as well as the circuit compilation correctness from QAFNY to SQIR (TR [24] C.6). We will further investigate integrating QAFNY with other tools, such as CoqQ [60], to verify HCQ programs automatically.

References

- 1 A. Ambainis. Quantum walk algorithm for element distinctness. In *45th Annual IEEE Symposium on Foundations of Computer Science*, pages 22–31, 2004. doi:[10.1109/FOCS.2004.54](https://doi.org/10.1109/FOCS.2004.54).
- 2 Stephane Beauregard. Circuit for shor’s algorithm using $2n+3$ qubits. *Quantum Info. Comput.*, 3(2):175–185, March 2003.
- 3 Christophe Charetton, Sébastien Bardin, François Bobot, Valentin Perrelle, and Benoît Valiron. An automated deductive verification framework for circuit-building quantum programs. In Nobuko Yoshida, editor, *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*, volume 12648 of *Lecture Notes in Computer Science*, pages 148–177. Springer, 2021. doi:[10.1007/978-3-030-72019-3_6](https://doi.org/10.1007/978-3-030-72019-3_6).
- 4 Andrew Childs, Ben Reichardt, Robert Spalek, and Shengyu Zhang. Every NAND formula of size N can be evaluated in time $N^{1/2+o(1)}$ on a Quantum Computer. *CoRR*, 2007. doi:[/10.48550/arXiv.quant-ph/0703015](https://doi.org/10.48550/arXiv.quant-ph/0703015).
- 5 Andrew M. Childs. On the Relationship Between Continuous- and Discrete-Time Quantum Walk. *Communications in Mathematical Physics*, 294(2):581–603, October 2009.
- 6 Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 23–42, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- 7 Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. Open quantum assembly language. *arXiv e-prints*, July 2017. arXiv:1707.03429.
- 8 Paul Adrien Maurice Dirac. A new notation for quantum mechanics. *Mathematical Proceedings of the Cambridge Philosophical Society*, 35:416–418, 1939.

- 9 Mahmudul Faisal Al Ameen and Makoto Tatsuta. Completeness for recursive procedures in separation logic. *Theoretical Computer Science*, 631:73–96, 2016. doi:[10.1016/j.tcs.2016.04.004](https://doi.org/10.1016/j.tcs.2016.04.004).
- 10 Yuan Feng and Mingsheng Ying. Quantum hoare logic with classical variables. *ACM Transactions on Quantum Computing*, 2(4), December 2021. doi:[10.1145/3456877](https://doi.org/10.1145/3456877).
- 11 Sukhpal Singh Gill, Oktay Cetinkaya, Stefano Marrone, Daniel Claudino, David Haunschild, Leon Schlotte, Huaming Wu, Carlo Ottaviani, Xiaoyuan Liu, Sree Pragna Machupalli, Kamalpreet Kaur, Priyansh Arora, Ji Liu, Ahmed Farouk, Houbing Herbert Song, Steve Uhlig, and Kotagiri Ramamohanarao. Quantum computing: Vision and challenges, 2024. [arXiv:2403.02240](https://arxiv.org/abs/2403.02240).
- 12 Daniel M. Greenberger, Michael A. Horne, and Anton Zeilinger. *Going beyond Bell's Theorem*, pages 69–72. Springer Netherlands, Dordrecht, 1989. doi:[10.1007/978-94-017-0849-4_10](https://doi.org/10.1007/978-94-017-0849-4_10).
- 13 Lov K. Grover. Quantum mechanics helps in searching for a needle in a haystack. *Phys. Rev. Lett.*, 79:325–328, July 1997. doi:[10.1103/PhysRevLett.79.325](https://doi.org/10.1103/PhysRevLett.79.325).
- 14 Thomas Häner, Martin Roetteler, and Krysta M. Svore. Factoring using $2n + 2$ qubits with toffoli based modular multiplication. *Quantum Info. Comput.*, 17(7–8):673–684, June 2017.
- 15 Kesha Hietala, Robert Rand, Shih-Han Hung, Liyi Li, and Michael Hicks. Proving quantum programs correct. In *Proceedings of the Conference on Interative Theorem Proving (ITP)*, June 2021.
- 16 Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. A verified optimizer for quantum circuits. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, January 2021.
- 17 C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969. doi:[10.1145/363235.363259](https://doi.org/10.1145/363235.363259).
- 18 Shachar Itzhaky, Hila Peleg, Nadia Polikarpova, Reuben N. S. Rowe, and Ilya Sergey. Cyclic program synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 944–959, New York, NY, USA, 2021. Association for Computing Machinery. doi:[10.1145/3453483.3454087](https://doi.org/10.1145/3453483.3454087).
- 19 Yoshihiko Kakutani. A logic for formal verification of quantum programs. In Anupam Datta, editor, *Advances in Computer Science - ASIAN 2009. Information Security and Privacy*, pages 79–93, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- 20 Xuan-Bach Le, Shang-Wei Lin, Jun Sun, and David Sanan. A quantum interpretation of separating conjunction for local reasoning of quantum programs based on separation logic. *Proc. ACM Program. Lang.*, 6(POPL), January 2022. doi:[10.1145/3498697](https://doi.org/10.1145/3498697).
- 21 K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- 22 K. Rustan M. Leino and Michał Moskal. Co-induction simply. In Cliff Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods*, pages 382–398, Cham, 2014. Springer International Publishing.
- 23 Liyi Li, Finn Voichick, Kesha Hietala, Yuxiang Peng, Xiaodi Wu, and Michael Hicks. Verified compilation of quantum oracles. In *OOPSLA 2022*, 2022. doi:[10.48550/arXiv.2112.06700](https://doi.org/10.48550/arXiv.2112.06700).
- 24 Liyi Li, Mingwei Zhu, Rance Cleaveland, Alexander Nicolellis, Yi Lee, Le Chang, and Xiaodi Wu. Qafny: A quantum-program verifier, 2024. [arXiv:2211.06411](https://arxiv.org/abs/2211.06411).
- 25 Yangjia Li and Dominique Unruh. Quantum Relational Hoare Logic with Expectations. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*, volume 198 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 136:1–136:20, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:[10.4230/LIPIcs.ICALP.2021.136](https://doi.org/10.4230/LIPIcs.ICALP.2021.136).
- 26 Junyi Liu, Bohua Zhan, Shuling Wang, Shenggang Ying, Tao Liu, Yangjia Li, Mingsheng Ying, and Naijun Zhan. Formal verification of quantum algorithms using quantum hoare logic. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 187–207, Cham, 2019. Springer International Publishing.

24:30 Qafny: A Quantum-Program Verifier

- 27 Christof Löding, P. Madhusudan, and Lucas Peña. Foundations for natural proofs and quantifier instantiation. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158098.
- 28 Guang Hao Low and Isaac L. Chuang. Optimal Hamiltonian Simulation by Quantum Signal Processing. *Physical Review Letters*, 118(1), January 2017.
- 29 Partha Sarathy Madhusudan, Xiaokang Qiu, and Andrei Stefanescu. Recursive proofs for inductive tree data-structures. *SIGPLAN Not.*, 47(1):123–136, January 2012. doi:10.1145/2103621.2103673.
- 30 Frédéric Magniez, Miklos Santha, and Mario Szegedy. Quantum Algorithms for the Triangle Problem. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '05, pages 1109–1117, USA, 2005. Society for Industrial and Applied Mathematics.
- 31 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.
- 32 Microsoft. Probabilistic Z3, 2014. URL: <https://github.com/ProbabilisticZ3/src>.
- 33 Yunseong Nam, Neil J. Ross, Yuan Su, Andrew M. Childs, and Dmitri Maslov. Automated optimization of large quantum circuits with continuous parameters. *npj Quantum Information*, 4(1):23, May 2018. doi:10.1038/s41534-018-0072-4.
- 34 Daniel Neider, Pranav Garg, P. Madhusudan, Shambwaditya Saha, and Daejun Park. Invariant synthesis for incomplete verification engines. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 232–250, Cham, 2018. Springer International Publishing.
- 35 Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, USA, 10th anniversary edition, 2011.
- 36 KARL PEARSON. The problem of the random walk. *Nature*, 72(1865):294–294, July 1905. doi:10.1038/072294b0.
- 37 Edgar Pek, Xiaokang Qiu, and P. Madhusudan. Natural proofs for data structure manipulation in c using separation logic. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 440–451, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2594291.2594325.
- 38 Yuxiang Peng, Kesha Hietala, Runzhou Tao, Liyi Li, Robert Rand, Michael Hicks, and Xiaodi Wu. A formally certified end-to-end implementation of Shor’s factorization algorithm. *Proceedings of the National Academy of Sciences*, 120(21):e2218775120, 2023. doi:10.1073/pnas.2218775120.
- 39 Xiaokang Qiu, Pranav Garg, Andrei Stefanescu, and Partha Sarathy Madhusudan. Natural proofs for structure, data, and separation. *SIGPLAN Not.*, 48(6):231–242, June 2013. doi:10.1145/2499370.2462169.
- 40 Xiaokang Qiu, Pranav Garg, Andrei Stefanescu, and Partha Sarathy Madhusudan. Natural proofs for structure, data, and separation. 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 231–242. ACM, 2013. doi:10.1145/2491956.2462169.
- 41 Robert Rand. *Formally verified quantum programming*. PhD thesis, University of Pennsylvania, 2018.
- 42 J.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, 2002. doi:10.1109/LICS.2002.1029817.
- 43 Gustavo Rigolin. Quantum teleportation of an arbitrary two-qubit state and its relation to multipartite entanglement. *Physical Review A*, 71(3), March 2005. doi:10.1103/physreva.71.032303.

- 44 Grigore Roșu and Andrei Ștefănescu. Matching Logic: A New Program Verification Approach (NIER Track). In *ICSE'11: Proceedings of the 30th International Conference on Software Engineering*, pages 868–871. ACM, 2011. doi:doi:10.1145/1985793.1985928.
- 45 Grigore Roșu, Andrei Ștefănescu, Ștefan Ciobâcă, and Brandon M. Moore. One-Path Reachability Logic. In *Proceedings of the 28th Symposium on Logic in Computer Science (LICS'13)*, pages 358–367. IEEE, June 2013.
- 46 Michael Sammeler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. Refinedc: Automating the foundational verification of c code with refined ownership types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 158–174, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3453483.3454036.
- 47 P. W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, FOCS '94, 1994.
- 48 P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994. doi:10.1109/SFCS.1994.365700.
- 49 Matt Swayne. What Are The Remaining Challenges Of Quantum Computing?, 2023. URL: <https://thequantuminsider.com/2023/03/24/quantum-computing-challenges/>.
- 50 Quang-Trung Ta, Ton Chanh Le, Siau-Cheng Khoo, and Wei-Ngan Chin. Automated mutual explicit induction proof in separation logic, 2016. doi:10.48550/arXiv.1609.00919.
- 51 Makoto Tatsuta, Wei-Ngan Chin, and Mahmudul Faisal Al Ameen. Completeness and expressiveness of pointer program verification by separation logic. *Information and Computation*, 267:1–27, 2019. doi:10.1016/j.ic.2019.03.002.
- 52 Dominique Unruh. Quantum relational hoare logic. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290346.
- 53 Salvador Elías Venegas-Andraca. Quantum walks: a comprehensive review. *Quantum Information Processing*, 11(5):1015–1106, July 2012. doi:10.1007/s11128-012-0432-5.
- 54 Thomas G. Wong. Unstructured search by random and quantum walk. *Quantum Information and Computation*, 22(1&2):53–85, January 2022. doi:10.26421/qic22.1-2-4.
- 55 Hongseok Yang and Peter O’Hearn. A semantic basis for local reasoning. In Mogens Nielsen and Uffe Engberg, editors, *Foundations of Software Science and Computation Structures*, pages 402–416, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.
- 56 Mingsheng Ying. Floyd–hoare logic for quantum programs. *ACM Trans. Program. Lang. Syst.*, 33(6), January 2012. doi:10.1145/2049706.2049708.
- 57 Mingsheng Ying. Toward automatic verification of quantum programs. *Form. Asp. Comput.*, 31(1):3–25, February 2019. doi:10.1007/s00165-018-0465-3.
- 58 Bohua Zhan. Efficient verification of imperative programs using auto2. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 23–40, Cham, 2018. Springer International Publishing.
- 59 Li Zhou, Gilles Barthe, Justin Hsu, Mingsheng Ying, and Nengkun Yu. A Quantum Interpretation of Bunched Logic and Quantum Separation Logic. In *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS ’21, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1109/LICS52264.2021.9470673.
- 60 Li Zhou, Gilles Barthe, Pierre-Yves Strub, Junyi Liu, and Mingsheng Ying. Coqq: Foundational verification of quantum programs. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. doi:10.1145/3571222.

Compositional Symbolic Execution for Correctness and Incorrectness Reasoning

Andreas Lööw

Imperial College London, UK

Daniele Nantes-Sobrinho

Imperial College London, UK

Sacha-Élie Ayoun

Imperial College London, UK

Caroline Cronjäger

Ruhr-Universität Bochum, Germany

Petar Maksimović

Imperial College London, UK

Runtime Verification Inc., Chicago, IL, USA

Philippa Gardner

Imperial College London, UK

Abstract

The introduction of separation logic has led to the development of symbolic execution techniques and tools that are (functionally) compositional with function specifications that can be used in broader calling contexts. Many of the compositional symbolic execution tools developed in academia and industry have been grounded on a formal foundation, but either the function specifications are not validated with respect to the underlying separation logic of the theory, or there is a large gulf between the theory and the implementation of the tool.

We introduce a formal compositional symbolic execution engine which creates and uses function specifications from an underlying separation logic and provides a sound theoretical foundation for, and indeed was partially inspired by, the Gillian symbolic execution platform. This is achieved by providing an axiomatic interface which describes the properties of the consume and produce operations used in the engine to update compositionally the symbolic state, for example when calling function specifications. This consume-produce technique is used by VeriFast, Viper, and Gillian, but has not been previously characterised independently of the tool. As part of our result, we give consume and produce operations inspired by the Gillian implementation that satisfy the properties described by our axiomatic interface. A surprising property is that our engine semantics provides a common foundation for both correctness and incorrectness reasoning, with the difference in the underlying engine only amounting to the choice to use satisfiability or validity. We use this property to extend the Gillian platform, which previously only supported correctness reasoning, with incorrectness reasoning and automatic true bug-finding using incorrectness bi-abduction. We evaluate our new Gillian platform by using the Gillian instantiation to C. This instantiation is the first tool grounded on a common formal compositional symbolic execution engine to support both correctness and incorrectness reasoning.

2012 ACM Subject Classification Theory of computation → Program verification; Theory of computation → Program analysis; Theory of computation → Separation logic; Theory of computation → Automated reasoning

Keywords and phrases separation logic, incorrectness logic, symbolic execution, bi-abduction

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.25

Related Version *Extended Version:* <https://doi.org/10.48550/arXiv.2407.10838> [18]

Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact):* <https://doi.org/10.4230/DARTS.10.2.13>

 © Andreas Lööw, Daniele Nantes-Sobrinho, Sacha-Élie Ayoun, Caroline Cronjäger, Petar Maksimović, and Philippa Gardner,
licensed under Creative Commons License CC-BY 4.0
38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 25; pp. 25:1–25:28



Funding This work was supported by the EPSRC Fellowship “VetSpec: Verified Trustworthy Software Specification” (EP/R034567/1).

Acknowledgements We would like to thank Nat Karmios for help with preparing the artefact for this paper. We would also like to thank the anonymous reviewers for their comments.

1 Introduction

One of the main challenges that modern program analysis tools based on static symbolic execution [1] must face is *scalability*, that is, the ability to tractably analyse large, dynamically changing codebases. Such scalability requires symbolic techniques and tools that are *functionally compositional* (or simply compositional) where the analysis works on functions in isolation, at any point in the codebase, and then records the results in simple function specifications that can be used in broader calling contexts. However, the traditional symbolic execution tools and frameworks based on first-order logic, such as CBMC [16] and Rosette [32], can only be compositional for functions that manipulate the variable store, but not for functions that manipulate the heap, limiting their usability.

A key insight is that, to obtain compositionality, the analysis should work with function specifications that are *local*, in that they should describe the function behaviour only on the *partial* states or resources that the function accesses or manipulates, and a mechanism for using such specifications when the function is called by code working on a larger state. This insight was first introduced in separation logic (SL) [24, 29], a modern over-approximating (OX) program logic for compositional verification of correctness properties, which features local function specifications that can be called on larger state with the help of the *frame rule*. Recently, these ideas have been adapted to under-approximate (UX) reasoning in the context of incorrectness separation logic (ISL) [28] for compositional true bug-finding.

Ideas from SL and ISL have led to the development of efficient compositional symbolic execution tools in academia and industry, such as the tool VeriFast [14] and the multi-language platforms Viper [23] and Gillian [21] for semi-automatic OX verification based on SL, and Meta’s multi-language platform Infer-Pulse [17] for automatic UX true bug-finding based on ISL. However, there are issues with the associated formalisms of the tools: either the function specifications created and used by the tool are not validated with respect to the underlying separation logic; or there is a large gulf between the formalism and the implementation of the tool. VeriFast, Viper, and Gillian, on the one hand, all come with a sound compositional symbolic operational semantics that closely model the tools. These tools handle the creation and use of function specifications (and the folding and unfolding of predicates) using two operations, called *consume* and *produce*, which, respectively, removes from and adds to a given symbolic state the symbolic state corresponding to a given assertion. The formalisms accompanying the tools, however, do not properly connect their function specifications to the underlying separation logics. Thus, function specifications created by the tools cannot soundly be used by other tools, and vice versa. On the other hand, the formalism of Infer-Pulse describes compositional symbolic execution as proof search in ISL, and similarly with its SL-predecessor Infer [4], thereby making the connection to its separation logic direct. However, the gap between the formalism and the tool is considerable.

In this paper, inspired by the Gillian platform [11, 21], we formally define a compositional symbolic execution (CSE) engine that provides a sound theoretical foundation for building compositional OX and UX analysis tools. Our engine is described by a compositional symbolic operational semantics using the consume and produce operations to interface with function

specifications valid in either SL or ISL. A surprising property of our semantics is that it is simple to switch between OX and UX reasoning. In more detail, our CSE engine features the following theoretical contributions:

1. *specification interoperability*, in the sense that it can both create and use function specifications validated in an underlying separation logic, allowing for a mix-and-match of specifications validated in various ways from various sources;
2. *an axiomatic approach to compositionality*, in that we provide an axiomatic description of properties for the consume and produce operations, which have not been previously characterised axiomatically;
3. *a general soundness result*, which states that, assuming the axiomatic properties of the consume and produce operations and the validity of function specifications that are used with respect to the underlying separation logic, the CSE engine is sound and creates function specifications that are valid with respect to the said logic;
4. *a unified semantics*, which captures the essence of *both* the OX reasoning of VeriFast, Viper, and Gillian, and the UX reasoning in Infer-Pulse, with the difference amounting *only* to the choice of using satisfiability or validity, and different axiomatic properties of the consume and produce operations.

We instantiate our general soundness result by giving example implementations of the consume and produce operations, inspired by those found in Gillian, which we prove satisfy the properties laid out by the axiomatic interface. For clarity of presentation, although both our CSE engine and our consume and produce operations are inspired by Gillian, we have opted to work with a fixed, linear memory model and a simple while language instead of the parametric memory model approach of Gillian and its goto-based intermediate language GIL. The move from a fixed to a parametric memory model is straightforward and planned future work.

In addition, this paper brings the following practical contributions:

1. a demonstrator Haskell implementation of our CSE engine with example implementations of the consume and produce operations;
2. an extension of the Gillian platform with automatic compositional UX true bug-finding using UX bi-abduction in the style of Infer-Pulse, making Gillian the first unified tool for OX and UX compositional reasoning about real-world code.

The Gillian platform already supported whole-program symbolic testing as found in, for example, CBMC, and semi-automatic OX compositional verification underpinned by SL as in, for example, VeriFast. Because our CSE engine has pinpointed the small differences required for the switch between OX and UX reasoning, we were able to simply extend Gillian with automatic compositional UX true-bug finding without affecting its other analyses. Interestingly, UX true bug-finding has not been implemented in a consume-produce engine before. We demonstrate the additional UX reasoning by extending the CSE engine with UX bi-abductive reasoning [5, 6, 28, 17], an automatic technique which has enabled compositional reasoning to scale to industry-grade codebases, and which works by generating function specifications from their implementations by incrementally constructing the calling context. We implement this technique following the approach pioneered by OX tool JaVerT 2.0 [10], where missing-resource errors are used to generate fixes that drive the specification construction. We evaluate this extension of Gillian using its Gillian-C instantiation, on a real-world Collections-C data-structure library [25], obtaining promising initial results and performance.

An extended version of this paper is available on arXiv [18], which includes additional definitions and proofs of the theorems discussed in this paper.

2 Overview: Compositional Symbolic Execution

We give an overview of our CSE engine, together with example analyses that we show can be hosted on top of this engine. Our CSE engine consists of three engines built on top of each other, labelled by different reasoning modes, OX and UX, that are appropriate for different types of analyses. In short, the reasoning modes can be characterised as follows:

Mode	Guarantee	Consequence rule	Analysis
OX	Full path coverage	Forward logical consequence	Full verification in SL
UX	Path reachability	Backward logical consequence	True bug-finding in ISL

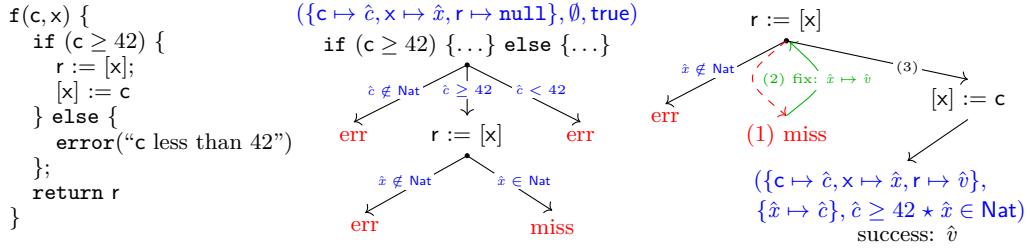
The *core engine* (§4) is a simple symbolic execution engine for our demonstrator programming language (§3). It does not use assertions to update symbolic state and is hence compatible with both OX and UX reasoning. It is sufficient to capture whole-program symbolic testing found in, e.g., CMBC and Gillian. The *compositional engine* (§5, §6) is built on top of this core engine. It can switch between either the OX or UX mode of reasoning, providing support for the use of SL and ISL function specifications by extending the core engine with consume and produce operations for updating the symbolic state. In OX mode, it captures the full verification found in e.g. VeriFast and Gillian, soundly underpinned by SL. For the first time, in UX mode it also captures ISL analysis, not previously found in a symbolic execution tool. We demonstrate this by building the UX *bi-abductive engine* (§7) on top of the UX compositional engine to automatically fix missing-resource errors (e.g., a missing heap cell) using the UX bi-abductive technique from Infer-Pulse, to capture automatic true bug-finding underpinned by ISL.

2.1 Core Engine

The core symbolic execution engine provides a foundation on top of which the other components are built. It is essentially a standard symbolic execution engine that is slightly adapted to handle both usual language errors and the missing-resource errors, which can occur now that we are working with partial state.

Our engine operates over partial symbolic states $\hat{\sigma} = (\hat{s}, \hat{h}, \hat{\pi})$ comprising: a symbolic variable store \hat{s} , holding symbolic values for the program variables; a partial symbolic heap \hat{h} , representing the memory on which programs operate; and a symbolic path condition $\hat{\pi}$, holding constraints accumulated during symbolic execution. We work with a simple demonstrator programming language (cf. §3) and linear heaps: that is, partial-finite maps from natural numbers to values. The core engine is both OX- and UX-sound, also referred to as exact (EX) [20], as established by Thm. 1.

Example. In Fig. 1 (left), we define a simple function f . In Fig. 1 (middle), we illustrate its symbolic execution, which starts from $\hat{\sigma} = (\{c \mapsto \hat{c}, x \mapsto \hat{x}, r \mapsto \text{null}\}, \emptyset, \text{true})$, with the function parameters (c and x) initialised with some symbolic variables (\hat{c} and \hat{x}), the local function variables (r) initialised to null , the heap set to empty and the path condition set to true . Next, executing the if-statement with condition $c \geq 42$ yields three branches: one in which c is not a natural number, in which the execution fails with an evaluation error; one in which $c \geq 42$, in which the execution continues; and one in which $c < 42$, in which the program throws a user-defined error. Next, executing the lookup $r := [x]$ results in two more branches: one in which x is not a heap address (natural number), yielding a type error and one in which x is a heap address. In that branch, the lookup fails with a missing-resource error as the heap is empty.



■ **Figure 1** Definition and symbolic execution of function f .

Analysis: EX Whole-program Symbolic Testing. The core engine can be used to perform whole-program symbolic testing in the style of CBMC [16] and Gillian [11], in which the user creates symbolic variables, imposes some initial constraints on them, runs the symbolic execution to completion, and asserts that some final constraints hold.

2.2 Compositional Engine

Our compositional engine extends the core engine to support calling, in its respective OX and UX mode, SL and ISL function specifications, denoted $\{\vec{x} = \vec{x} \star P\} f(\vec{x}) \{ok : Q_{ok}\} \{err : Q_{err}\}$ and $[\vec{x} = \vec{x} \star P] f(\vec{x}) [ok : Q_{ok}] [err : Q_{err}]$,¹ respectively, where P is a pre-condition and Q_{ok} and Q_{err} are success and error post-conditions. A SL specification gives an OX description of the function behaviour whereas an ISL specification gives a UX description:

- (SL) All terminating executions of the function f starting in a state satisfying P either end successfully in a state that satisfies Q_{ok} or fault in a state that satisfies Q_{err} .
- (ISL) Any state satisfying either the success Q_{ok} or error post-conditions Q_{err} is reachable from some state satisfying the pre-condition P by executing the function f .

The engine also adds support, in both OX and UX mode, for folding and unfolding of user-defined predicates, describing inductive data-structures such as linked lists.

In both cases, the call to function specifications and the folding and unfolding of predicates are implemented following the consume-produce engine style, underpinned by `consume` and `produce` operations, which, in essence, remove (consume) and add (produce) the symbolic state corresponding to a given assertion from and to the current symbolic state. For example, in Fig. 2, the symbolic execution is in a symbolic state $\hat{\sigma}$ and calls a function $f(\vec{x})$ by its specification in ISL mode. The (successful) function call is implemented by first consuming the symbolic state $\hat{\sigma}_P$ corresponding to the pre-condition P , leaving the symbolic frame $\hat{\sigma}_f$, and then producing into $\hat{\sigma}_f$ the symbolic state $\hat{\sigma}_{Q_{ok}}$ corresponding to the post-condition Q_{ok} .

Our approach is novel in two ways: (1) we provide an axiomatic interface that captures the sufficient properties of the consume and produce operations for the engine to be sound; and (2) we provide example implementations (in the same style as the rest of the engine, that is, using inference rules) for the consume and produce operations that we prove satisfy the axiomatic interface. Moreover, our consume and produce operations switch their behaviour between the mode of reasoning (OX/UX), as described next.

¹ UX quadruples can be split into two triples, but OX quadruples cannot. To unify our presentation, we consider both types of specifications in quadruple form.

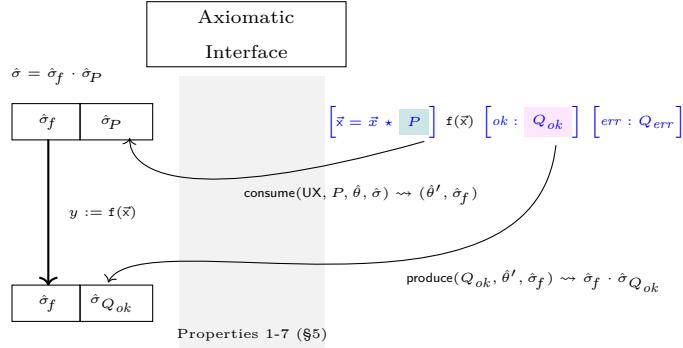


Figure 2 UX function-call rule: successful case.

Axiomatic Interface. We have identified sufficient properties for the consume and produce operations to be OX or UX sound (cf. Thm. 3). Here we will describe a general idea of the consume operation, the more complex of the two operations, the signature of which is:

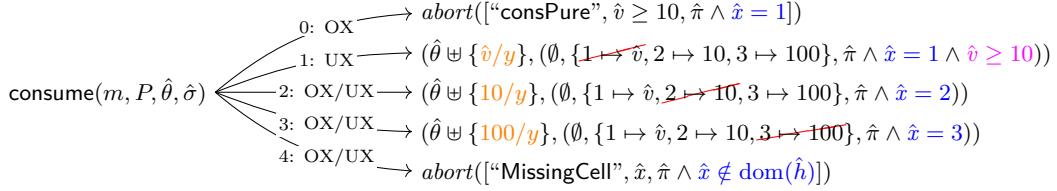
$$\text{consume}(m, P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow (\hat{\theta}', \hat{\sigma}_f) \mid \text{abort}(\hat{v})$$

The consume operation takes a mode m (OX or UX), an assertion P , a substitution $\hat{\theta}$, and a symbolic state $\hat{\sigma}$, where the substitution $\hat{\theta}$ links known logical variables in P to symbolic values in $\hat{\sigma}$. The operation finds which part of $\hat{\sigma}$ could match P , resulting in potentially multiple successful or unsuccessful matches, and then, per match, returns either the pair $(\hat{\theta}', \hat{\sigma}_f)$, which comprises a substitution $\hat{\theta}'$ and a resulting symbolic state $\hat{\sigma}_f$, or it aborts with error information \hat{v} . Some properties the interface of `consume` mandates are the following:

1. In successful consumption, there exists a symbolic state $\hat{\sigma}_P$ such that $\hat{\sigma} = \hat{\sigma}_f \cdot \hat{\sigma}_P$ (where \cdot denotes state composition, which composes the corresponding components of the two states together) and that every concrete state described by $\hat{\sigma}_P$ satisfies P . This tells us that the matched part of $\hat{\sigma}$ does correspond to P , that the effect of `consume` is its removal from $\hat{\sigma}$, and that consumption can be viewed as the frame inference problem [3], with the resulting state $\hat{\sigma}_f$ constituting the frame;
2. In OX mode, `consume` does not drop paths; in UX mode, it does not drop information. The interface allows, e.g., tool developers to design an OX-consume that (soundly) drops certain information deemed unneeded or a UX-consume that (soundly) drops paths according to a tool-specific criteria (e.g., as in UX bi-abduction in Infer-Pulse).

Example Implementation: Consume. We provide example implementations for the consume and produce operations (§6) that explore the similarities between OX and UX reasoning, and allow us to maintain unified implementations across both reasoning modes. Our consume operation has a mode switch m , allowing for OX- or UX-specific behaviour, which we use to control the *only difference* in our implementation between the two modes: the consumption of pure (non-spatial) information (cf. Fig. 7, left). For soundness, our implementation of the consume operation has to be compatible with the SL and ISL guarantees: in OX mode, `consume` requires full path coverage, and in UX, it requires path reachability.

We illustrate our `consume` implementation by example. Consider the symbolic state $\hat{\sigma} \triangleq (\emptyset, \hat{h}, \hat{\pi})$, where $\hat{h} \triangleq \{1 \mapsto \hat{v}, 2 \mapsto 10, 3 \mapsto 100\}$ and $\hat{\pi} \triangleq \hat{x} > 0 \wedge \hat{v} > 5$, and let us consume the assertion $P \triangleq x \mapsto y * y \geq 10$ from $\hat{\sigma}$ knowing that $\hat{\theta} = \{\hat{x}/x\}$, meaning that the logical variable x of P is mapped to the symbolic variable \hat{x} of $\hat{\sigma}$. This consumption is presented in the diagram below:



- the arrows are labelled with the mode m of operation of `consume`, being either only OX, or only UX, or OX/UX when the consumption has the same behaviour in both modes;
- both our OX and UX consumption branch on all possible matches: in this case, the cell assertion $x \mapsto y$ can be matched to any of the three cells in the heap (branches 0–3), but it could also refer to a cell that is outside of the heap (branch 4);
- when branching occurs, then the branching condition is added to the path condition of the resulting state (the constraints in blue), ensuring information is not dropped;
- the heap cell corresponding to $x \mapsto y$ is removed when matched successfully (branches 1, 2, 3), and in those cases we learn the value corresponding to y (substitution extension in orange where \uplus denotes disjoint union);
- for the heap cell $\{1 \mapsto \hat{v}\}$, our OX consumption (branch 0) must abort since the $\hat{\pi}$ does not imply $y \geq 10$ when $y = \hat{v}$, whereas UX consumption (branch 1) can proceed by restricting the path condition (constraint in magenta), since dropping paths is sound in UX; this allows our UX consumption to successfully consume more assertions; OX consumption cannot do the same since that would render e.g. the function-call rule, which is implemented in terms of `consume`, unsound in OX mode;
- our UX consumption could alternatively drop the missing-cell abort outcome (branch 4), however, some analyses, such as bi-abduction, have use for this error information so we do not drop it.

Analysis: Verification. We use our compositional engine to provide semi-automatic OX verification: that is, given a function implementation and an OX function specification, the engine checks if the implementation satisfies the specification. This analysis is semi-automatic in that the user may have to provide loop invariants as well as ghost commands for, e.g., predicate manipulation and lemma application. It is implemented in the standard way for consume-produce tools.

2.3 Bi-abductive Engine

Bi-abduction is a technique that enables automatic compositional analysis by allowing incremental discovery of resources needed to execute a given piece of code. It was introduced in the OX verification setting [5, 6], later forming the basis of the bug-finding tool Infer [4], and was recently ported to the UX setting of true bug-finding, underpinning Infer-Pulse [17].

Our UX bi-abduction advances the state of the art in two ways. Firstly, UX bi-abduction has thus far been intertwined with the proof search of the symbolic execution tool it has formulated for [28, 17]. Inspired by an approach developed in the OX tool JaVerT 2.0 [10], we add UX bi-abduction as a layer on top of CSE by generating fixes from missing-resource errors. This covers both missing-resource errors from the execution of the commands of the language (e.g., in heap lookup, the looked-up cell might not be in the heap) as well as invocations of `consume` (e.g., if the resource required by a function pre-condition is not in the heap). In more detail, when a missing-resource error occurs, a fix is generated and applied to

the current symbolic state, allowing the execution to continue. Secondly, our UX bi-abduction is able to reason about predicates, allowing us to synthesise and soundly use a broader range of function specifications from other formalisms and tools, in particular specifications that capture unbounded behaviour rather than bounded or single-path behaviour.

Analysis: Specification Synthesis and True Bug-finding. We use bi-abduction to power automatic synthesis of UX function specifications, obtaining one specification per each constructed execution path. Such function specification synthesis forms the back-end of Pulse-style true bug-finding, where specifications describing erroneous executions, after appropriate filtering, can be reported as bugs. Given the guarantees of UX reasoning, any bug (represented by a synthesised erroneous function specification) found during this process will be a true bug.

Example: Specification Synthesis. We illustrate how bi-abduction can be used for the synthesis of UX function specifications, using again the simple function $f(c, x)$ from Fig. 1 (left). The first and the third branches of Fig. 1 (middle) yield the following specifications:

$$\begin{aligned} [c = c \star x = x] \ f(c, x) \ [err : err = ["ExprEval", "c \geq 42"] \star c \notin \text{Nat}] \\ [c = c \star x = x] \ f(c, x) \ [err : err = ["Error", "c \text{ less than } 42"] \star c < 42] \end{aligned}$$

noting that information about local variables is discarded, the error value is returned in the dedicated program variable err , and symbolic variables are replaced with logical variables.

Using bi-abduction, the second branch of Fig. 1 (middle) now becomes Fig. 1 (right). The second branch of Fig. 1 (middle) has one branch in which x is not a heap address (natural number), yielding a type error and the following specification

$$[c = c \star x = x] \ f(c, x) \ [err : err = ["Type", "x", x, "Nat"] \star c \geq 42 \star x \notin \text{Nat}]$$

and one branch in which x is a heap address. In that branch, the lookup fails with a missing-resource error as the heap is empty, but in bi-abductive execution, that is, Fig. 1 (right), instead of failing we first generate the fix $\hat{x} \mapsto \hat{v}$, where \hat{v} is a fresh symbolic variable, and then add it to the heap and re-execute the lookup, which now succeeds. The rest of the function is executed without branching or errors, the function terminates and returns the value of r , which is \hat{v} . This branch results in the following specification:

$$[c = c \star x = x \star x \mapsto v] \ f(c, x) \ [ok : x \mapsto c \star c \geq 42 \star \text{ret} = v]$$

which illustrates an essential principle of bi-abduction, which is to add the fixes applied during execution (also known as *anti-frame*, highlighted in red) to the specification pre-condition.

Example: Specification Synthesis with Predicates. To exemplify how predicates can be useful during specification synthesis, consider the following variant of the standard singly-linked list predicate: $\text{list}(x; xs, vs)$, where x denotes the starting address of the list, and xs and vs denote node addresses and node values, respectively.² Both addresses and values are exposed in the predicate to ensure that no information is lost when the predicate is folded, making it suitable for UX reasoning. The predicate is defined as follows:

$$\begin{aligned} \text{list}(x; xs, vs) \triangleq (x = \text{null} \star xs = [] \star vs = []) \vee \\ (\exists v, x', xs', vs'. x \mapsto v, x' \star \text{list}(x'; xs', vs') \star xs = x : xs' \star vs = v : vs') \end{aligned}$$

² We use the semicolon notation for predicates to be consistent with the main text, where the notation is used for automation – for the purpose of this section, these semicolons can be read as commas.

Further, consider the predicate $\text{listHead}(x; xs)$, which tells us that x is the head of the list xs if xs is not empty and null otherwise, defined as

$$\text{listHead}(x; xs) \triangleq (xs = [] \star x = \text{null}) \vee (\exists xs'. xs = x : xs'),$$

and the following specifications of two list-manipulating functions (e.g., proven using pen-and-paper), which capture the exact behaviour of inserting a value in the front of a list (LInsert) and swapping of the first two values in a list (LSwapFirstTwo):

$$\begin{aligned} [x = x \star v = v \star \text{list}(x; xs, vs)] \text{ LInsert}(x, v) & [ok : \text{list}(\text{ret}; \text{ret}:xs, v:vs) \star \text{listHead}(x; xs)] \\ [x = x \star \text{list}(x; xs, vs)] \text{ LSwapFirstTwo}(x) & [err : \text{list}(x; xs, vs) \star |vs| < 2 \star \text{err} = \text{"List too short!"}] \end{aligned}$$

Using these specifications, we can bi-abduce the following UX true-bug specification of client code calling these functions, where the discovered anti-frame is again highlighted in red, but this time contains a predicate:

$$\begin{aligned} & [x = x \star \text{list}(x; xs, vs)] \\ & x := \text{LInsert}(x, 42); y := \text{LSwapFirstTwo}(x) \\ & [err : \exists x'. \text{list}(x'; x':xs, 42:vs) \star \text{listHead}(x; xs) \star |42:vs| < 2 \star \text{err} = \text{"List too short!"}] \end{aligned}$$

3 Programming Language

We present a simple imperative heap language with function calls on which our analysis engine operates. The language is standard, except that, in line with previous work on compositional reasoning and incorrectness [8, 9, 10, 12, 28], we track freed cells in the heap, and separate language errors and missing-resource errors to preserve the compositionality of the semantics. We sometimes refer to the definitions of this section as *concrete* to differentiate them from the *symbolic* definitions used in the symbolic engine introduced in subsequent sections.

Syntax. The values are given by: $v \in \text{Val} ::= n \in \text{Nat} \mid b \in \text{Bool} \mid s \in \text{Str} \mid \text{null} \mid [\vec{v}]$, where \vec{v} denotes a vector of values. The types are given by: $\tau \in \text{Type} ::= \text{Val} \mid \text{Nat} \mid \text{Bool} \mid \text{Str} \mid \text{List}$. The types are used to define the semantics of the language; the language itself is dynamically typed. The expressions, $E \in \text{PExp}$, comprise the values, program variables $x, y, z, \dots \in \text{PVar}$, and expressions formed using the standard operators for numerical and Boolean expressions. The commands are given by the grammar:

$$\begin{aligned} C \in \text{Cmd} ::= & \text{skip} \mid x := E \mid x := \text{nondet} \mid \text{error}(E) \mid x := [E] \mid [E] := E \mid x := \text{new} \mid \\ & \text{free}(E) \mid \text{if } (E) C \text{ else } C \mid C; C \mid y := f(\vec{E}) \end{aligned}$$

comprising the variable assignment, variable assignment of a non-deterministically chosen natural number, user-thrown error, heap lookup, heap mutation, allocation, deallocation, command sequencing, conditional control-flow and function call. Our results extend to other control-flow commands, e.g. loops, since these can be implemented using conditionals and recursive functions. The sets of program variables for expressions $\text{pv}(E)$ and commands $\text{pv}(C)$ are standard.

Functions and Function Implementation Contexts. A function implementation, denoted $f(\vec{x}) \{ C; \text{return } E \}$, comprises: an identifier, $f \in \text{Fid} \subseteq \text{Str}$; the parameters, given by a list of distinct program variables \vec{x} ; a body, $C \in \text{Cmd}$; and a return expression, $E \in \text{PExp}$, with $\text{pv}(E) \subseteq \{\vec{x}\} \cup \text{pv}(C)$. A function implementation context, γ , maps function identifiers to their implementations: $\gamma : \text{Fid} \rightarrow_{\text{fin}} \text{PVar List} \times \text{Cmd} \times \text{PExp}$, where \rightarrow_{fin} denotes that the function is a finite partial function. We often write $f(\vec{x}) \{ C; \text{return } E \} \in \gamma$ for $\gamma(f) = (\vec{x}, C, E)$.

Stores, Heaps and States. A variable store, $s : \text{PVar} \rightarrow_{\text{fin}} \text{Val}$, is a function from program variables to values. A *partial* heap, $h : \text{Nat} \rightarrow_{\text{fin}} (\text{Val} \uplus \{\emptyset\})$, is a function from natural numbers to values extended with a dedicated symbol $\emptyset \notin \text{Val}$ recording that a heap cell has been freed. Two heaps are disjoint, denoted $h_1 \# h_2$, when their domains are disjoint. Heap composition, denoted $h_1 \uplus h_2$, is given by the disjoint union of partial functions which is only defined when the domains are disjoint. A *partial* program state, $\sigma = (s, h)$, is a pair comprising a store and a heap. State composition, denoted $\sigma_1 \cdot \sigma_2$, is given by $(s_1, h_1) \cdot (s_2, h_2) \triangleq (s_1 \cup s_2, h_1 \uplus h_2)$ for $\sigma_1 = (s_1, h_1)$ and $\sigma_2 = (s_2, h_2)$, which is only defined when variable stores are equal on their intersection and the heap composition is defined.

Operational Semantics. We use a standard expression evaluation function, $\llbracket E \rrbracket_s$, which evaluates an expression E with respect to a store s , assuming that expressions do not affect the heap. It results in either a value or a dedicated symbol $\notin \text{Val}$, denoting, an evaluation error, such as a variable not being in the store or a mathematical error. The operational semantics of commands is a big-step semantics using judgements of the form $\sigma, C \Downarrow_\gamma o : \sigma'$ which reads “the execution of command C in state σ and function implementation context γ results in a state σ' with outcome o ”, where $o ::= \text{ok} \mid \text{err} \mid \text{miss}$ denotes, respectively, a successful execution, a language error, and a missing-resource error due to the absence of a required cell in the partial heap. The separation of the missing-resource errors from the language errors is important for compositional reasoning, since the language satisfies both the standard OX and UX frame properties when the outcome is not missing. The semantics is standard and given in full in [18], along with the frame properties it satisfies.

4 Compositional Symbolic Execution: Core Engine

We present our CSE engine in two stages. In this section, we present the core CSE engine, given by a standard compositional symbolic operational semantics presented here to establish notation and introduce key concepts to the non-specialist reader: the definitions are similar to those for whole-program symbolic execution; the difference is with the use of partial state which has the effect that we have the distinction between language errors and missing-resource errors. In §5.3, we extend the core engine with our semantic rules for function calls and folding/unfolding predicates, using an axiomatic description of the consume and produce operations given in §5.2.

4.1 Symbolic States

Let $S\text{Var}$ be a set of symbolic variables, disjoint from the set of program variables, PVar , and values, Val . Symbolic values are defined as follows:

$$\hat{v} \in S\text{Val} ::= v \mid \hat{x} \mid \hat{v} + \hat{v} \mid \dots \mid \hat{v} = \hat{v} \mid \neg \hat{v} \mid \hat{v} \wedge \hat{v} \mid \hat{v} \in \tau$$

A symbolic store, $\hat{s} : \text{PVar} \rightarrow_{\text{fin}} S\text{Val}$, is a function from program variables to symbolic values. A partial symbolic heap, $\hat{h} : S\text{Val} \rightarrow_{\text{fin}} (S\text{Val} \uplus \{\emptyset\})$, is a function from symbolic values to symbolic values extended with \emptyset . A path condition, $\hat{\pi} \in S\text{Val}$, is a symbolic Boolean expression that captures constraints imposed on symbolic variables during execution. A (partial) symbolic state is a triple of the form $\hat{\sigma} = (\hat{s}, \hat{h}, \hat{\pi})$. Throughout the paper, we only work with well-formed states $\hat{\sigma}$, denoted $\mathcal{Wf}(\hat{\sigma})$, the definition is uninformative (it ensures, e.g., that the addresses of the symbolic heap are interpreted as natural numbers), see [18]. We write $\hat{\sigma}.\text{pc}$ and $\hat{\sigma}[\text{pc} := \hat{\pi}']$ to denote, respectively, access and update the state path condition. We write $\text{sv}(X)$ to denote the set of symbolic variables of a given construct X : e.g., $\text{sv}(\hat{s})$ for symbolic stores, $\text{sv}(\hat{h})$ for symbolic heaps, etc.

$$\begin{array}{c}
\text{LOOKUP} \\
\frac{\begin{array}{c} \llbracket E \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow (\hat{v}, \hat{\pi}') \quad \hat{h}(\hat{v}_l) = \hat{v}_m \\ \hat{\pi}'' \triangleq (\hat{v}_l = \hat{v}) \wedge \hat{\pi}' \quad \text{SAT}(\hat{\pi}'') \end{array}}{(\hat{s}, \hat{h}, \hat{\pi}), x := [E] \Downarrow_{\Gamma} ok : (\hat{s}[x \mapsto \hat{v}_m], \hat{h}, \hat{\pi}'')} \\
\\
\text{LOOKUP-ERR-VAL} \\
\frac{\begin{array}{c} \llbracket E \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow (\hat{v}, \hat{\pi}') \\ \hat{v}_{err} \triangleq [\text{"ExprEval"}, \text{str}(E)] \end{array}}{(\hat{s}, \hat{h}, \hat{\pi}), x := [E] \Downarrow_{\Gamma} err : (\hat{s}_{err}, \hat{h}, \hat{\pi}')} \\
\\
\text{LOOKUP-ERR-MISSING} \\
\frac{\begin{array}{c} \llbracket E \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow (\hat{v}, \hat{\pi}') \quad \text{SAT}(\hat{\pi}') \wedge \hat{v} \in \text{Nat} \wedge \hat{v} \notin \text{dom}(\hat{h}) \\ \hat{v}_{err} \triangleq [\text{"MissingCell"}, \text{str}(E), \hat{v}] \end{array}}{(\hat{s}, \hat{h}, \hat{\pi}), x := [E] \Downarrow_{\Gamma} miss : (\hat{s}_{err}, \hat{h}, \hat{\pi}' \wedge \hat{v} \in \text{Nat} \wedge \hat{v} \notin \text{dom}(\hat{h}))}
\end{array}$$

■ **Figure 3** Excerpt of symbolic execution rules, where $\hat{s}_{err} = \hat{s}[\text{err} \mapsto v_{err}]$.

Symbolic Interpretations. A symbolic interpretation, $\varepsilon : \text{SVar} \rightarrow_{fin} \text{Val}$ maps symbolic variables to concrete values, and is used to define the meaning of symbolic states and state the soundness results of the engine. We lift interpretations to symbolic values, $\varepsilon : \text{SVal} \rightarrow_{fin} \text{Val}$, with the property that it is undefined if the resulting concrete evaluation faults. Satisfiability of symbolic values is defined as usual, i.e., $\text{SAT}(\hat{\pi}) \triangleq \exists \varepsilon. \varepsilon(\hat{\pi}) = \text{true}$. We further lift symbolic interpretations to stores, heaps, and states, overloading the ε notation.

4.2 Core Engine

The symbolic expression evaluation relation, $\llbracket E \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow (\hat{w}, \hat{\pi}')$, evaluates a program expression E with respect to a symbolic store \hat{s} and path condition $\hat{\pi}$. It results in either a symbolic value or an evaluation error, $\hat{w} \triangleq \hat{v} \mid \not{f}$, and a satisfiable path condition $\hat{\pi}' \Rightarrow \pi$, which may contain additional constraints arising from the evaluation (e.g., to prevent branching on division by zero). The core CSE semantics is described using the usual single-trace semantic judgement (below, left) which is used to state UX properties. It also induces the collecting semantic judgement (below, right), which is used to state OX properties.

$$\hat{\sigma}, C \Downarrow_{\gamma} o : \hat{\sigma}' \quad \hat{\sigma}, C \Downarrow_{\gamma} \hat{\Sigma}' \iff \hat{\Sigma}' = \{(o, \hat{\sigma}') \mid \hat{\sigma}, C \Downarrow_{\gamma} o : \hat{\sigma}'\}$$

We give the lookup rules for illustration in Fig. 3: for example, the rule LOOKUP branches over all possible addresses in the heap that can match the given address.

Our CSE semantics is both OX- and UX-sound, which we call exact: OX soundness captures that no paths are dropped by stating that the symbolic semantics includes all behaviour w.r.t. the concrete semantics; UX soundness captures that no information is dropped by stating that the symbolic semantics does not add behaviour w.r.t. the concrete semantics.

► **Theorem 1** (OX and UX soundness).

$$\begin{array}{ll}
(OX) & \hat{\sigma}, C \Downarrow_{\gamma} \hat{\Sigma}' \wedge \varepsilon(\hat{\sigma}), C \Downarrow_{\gamma} o : \sigma' \implies \exists \hat{\sigma}', \varepsilon' \geq \varepsilon. (o, \hat{\sigma}') \in \hat{\Sigma}' \wedge \sigma' = \varepsilon'(\hat{\sigma}') \\
(UX) & \hat{\sigma}, C \Downarrow_{\gamma} o : \hat{\sigma}' \wedge \varepsilon(\hat{\sigma}') = \sigma' \implies \varepsilon(\hat{\sigma}), C \Downarrow_{\gamma} o : \sigma'
\end{array}$$

where $\varepsilon' \geq \varepsilon$ denotes that ε' extends ε , i.e., $\varepsilon'(\hat{x}) = \varepsilon(\hat{x})$ for all $\hat{x} \in \text{dom}(\varepsilon)$.

5 Compositional Symbolic Execution: Full Engine

Our core CSE engine is limited in that it does not call function specifications written in a program logic, and it cannot fold and unfold user-defined predicates to verify, e.g., specifications of list algorithms. What is missing is a general description of how to update symbolic state using assertions from the function specifications and predicate definitions. In

VeriFast, Viper and Gillian, this symbolic-state update is given by their implementations of the consume and produce operations. We instead give an *axiomatic interface* for describing such symbolic-state update by providing a general characterisation of these consume and produce operations (§5.2). Using this interface, we are able to give general definitions of the function-call rule (§5.3) and the folding and unfolding of predicates that are independent of the underlying tool implementation. Assuming the appropriate properties stated in the axiomatic interface, we prove that the resulting CSE engine is either OX sound or UX sound. Moreover, because the axiomatic interface relates the behaviour of the consume and produce operations to the standard satisfaction relation of SL and ISL, our function-call rule is able to use any function specification proven correct with respect to the standard function specification validity of SL and ISL, including functions specification proven outside our engine. In the next section (§6), we demonstrate that the Gillian implementation of the consume and produce operations are correct with respect to our axiomatic interface.³

5.1 Assertions and Extended Symbolic States

We present assertions suitable for both SL and ISL reasoning, and also extend our symbolic states to account for predicates. It is helpful to make a clear distinction between the logical assertions and symbolic states: since we work with the linear heap, the gap between assertions and symbolic states is quite small; with more complex memory models and optimised symbolic representations of memory, the gap is larger and this distinction becomes essential.

Assertions. Let $x, y, z \in \text{LVar}$ denote logical variables, distinct from program and symbolic variables. The set of logical expressions, $E \in \text{LExp}$, extends program expressions to include logical variables. We work with the following set of assertions (other assertions are derivable):

$$\begin{aligned} \pi \in \text{BAsrt} &\triangleq E \mid E \in \tau \mid \neg\pi \mid \dots \mid \pi_1 \wedge \pi_2 \\ P \in \text{Asrt} &\triangleq \pi \mid \text{False} \mid P_1 \Rightarrow P_2 \mid P_1 \vee P_2 \mid \exists x. P \mid \\ &\quad \text{emp} \mid E_1 \mapsto E_2 \mid E \mapsto \emptyset \mid P_1 \star P_2 \mid p(\vec{E}_1; \vec{E}_2) \end{aligned}$$

where $E, E_1, E_2 \in \text{LExp}$, $x \in \text{LVar}$, and $p \in \text{Str}$. The assertions should by now be familiar from separation logic. They comprise the lift of the usual first-order Boolean assertions π , assertions built from the usual first-order connectives and quantifiers, and assertions well-known from separation logic: the empty assertion `emp`, the cell assertion $E_1 \mapsto E_2$ describing a heap cell at an address given by E_1 with value given by E_2 , the less well-known assertion $E \mapsto \emptyset$ describing a heap cell at address E that has been freed, the separating conjunction $P_1 \star P_2$, and the predicate assertions $p(\vec{E}_1; \vec{E}_2)$. The parameters of predicate assertions $p(\vec{E}_1; \vec{E}_2)$ are separated into in-parameters \vec{E}_1 (“ins”) and out-parameters \vec{E}_2 (“outs”) for automation purposes, as we discuss in §6; this separation does not affect the logical meaning of the predicate assertions. We write $\text{1v}(X)$ to denote free logical variables of a construct X : e.g., $\text{1v}(E)$ for logical expressions, $\text{1v}(P)$ for assertions, etc. We say that an assertion P is *simple* if it does not syntactically feature the separating conjunction; simple assertions are used in the definition of matching plans (§6.1).

Predicates. Predicate definitions are given by a set Preds containing elements of type $\text{Str} \times \text{LVar} \times \text{LVar} \times \text{Asrt}$, with the notation $p(\vec{x}_{\text{in}}; \vec{x}_{\text{out}}) \{P\} \in \text{Preds}$, where the string p denotes the predicate name, the lists of disjoint parameters $\vec{x}_{\text{in}}, \vec{x}_{\text{out}}$ denote the predicate

³ To our best understanding, there is a large overlap between Gillian’s consume and produce operations and those of Viper and VeriFast. We therefore expect them to also satisfy the OX properties of our interface (we have however not proven this fact).

ins and outs respectively, and the assertion $P = \bigvee_i (\exists \vec{y}_i. P_i)$ denotes the predicate body, which does not contain program variables and whose free logical variables are contained in $\{\vec{x}_{\text{in}}\} \cup \{\vec{x}_{\text{out}}\}$ which are disjoint from the bound variables \vec{y}_i , and where the P_i 's (denoting the assertions of the predicate definition) do not contain disjunctions or existential quantifiers.

Satisfiability. The meaning of an assertion P is defined by capturing the models of P using the standard satisfaction relation $\theta, \sigma \models P$ where $\theta : \text{LVar} \rightarrow_{\text{fin}} \text{Val}$ is a logical interpretation represented by a function from logical variables to values and σ is a program state (as defined in §3). The formal definition is included in the extended version of this paper [18].

Function Specifications. The quadruples $\langle \vec{x} = \vec{x} \star P \rangle f(\vec{x}) \langle \text{ok} : Q_{\text{ok}} \rangle \langle \text{err} : Q_{\text{err}} \rangle$ and $\langle \vec{x} = \vec{x} \star P \rangle f(\vec{x}) \langle \text{ok} : Q_{\text{ok}} \rangle \langle \text{err} : Q_{\text{err}} \rangle$ denote, respectively, a SL and an ISL function specification, as explained in §2.2. We write $\langle \vec{x} = \vec{x} \star P \rangle f(\vec{x}) \langle \text{ok} : Q_{\text{ok}} \rangle \langle \text{err} : Q_{\text{err}} \rangle$ to refer to either. Both quadruples record successful executions and language errors. They are unable to record missing-resource errors, as these errors do not satisfy the OX and UX frame properties. Missing errors can be removed automatically via UX bi-abduction (see §7).

Formally, we define function specifications using internalisation [20]. In short, internalisation relates internal specifications, which describe the internal behaviour of functions, to external specifications, which describe the external behaviour of functions. Internalisation is needed for ISL to allow the logic to drop information about function-local program variables at function boundaries, since dropping information is in general not allowed in ISL. The full definitions of function specifications and internalisation are included in [18].

A function specification context, $\Gamma \in \text{Fid} \rightarrow_{\text{fin}} \mathcal{P}(\mathcal{E}\text{Spec})$, maps function identifiers to a finite set of external specifications $\mathcal{E}\text{Spec}$. To simplify the presentation of the paper, we assume existential quantifiers only occur at the top level of external specifications. We denote the validity of Γ with respect to γ by $\models (\gamma, \Gamma)$, and validity of a function specification with respect to Γ by $\Gamma \models \langle \vec{x} = \vec{x} \star P \rangle f(\vec{x}) \langle \text{ok} : Q_{\text{ok}} \rangle \langle \text{err} : Q_{\text{err}} \rangle$.

Extended Symbolic States. To reason about unbounded execution to verify, for example, specifications of list algorithms, we extend the partial symbolic states defined in §4.1 with symbolic predicates of the form $p(\vec{v}_1; \vec{v}_2)$, with $p \in \text{Str}$ and $\vec{v}_1, \vec{v}_2 \in \text{SVal}$. An extended symbolic state \hat{s} is a tuple $(\hat{s}, \hat{H}, \hat{\pi})$ comprising a partial symbolic state \hat{s} , a symbolic resource $\hat{H} = (\hat{h}, \hat{\mathcal{P}})$ with symbolic heap \hat{h} and multiset of symbolic predicates $\hat{\mathcal{P}}$, and a symbolic path condition $\hat{\pi}$. Definitions of well-formedness of symbolic state and symbolic interpretations are extended as expected. We define $\varepsilon, \sigma \models (\hat{s}, \hat{H}, \hat{\pi})$ analogously to assertion satisfaction since the interpretation of a symbolic state with respect to symbolic interpretation $\varepsilon : \text{SVar} \rightarrow_{\text{fin}} \text{Val}$ is a relation and not a function (cf. §4) due to the presence of symbolic predicates.

The composition of two extended symbolic states is defined by:

$$(\hat{s}_1, \hat{H}_1, \hat{\pi}_1) \cdot (\hat{s}_2, \hat{H}_2, \hat{\pi}_2) \triangleq (\hat{s}_1 \cup \hat{s}_2, \hat{H}_1 \cup \hat{H}_2, \hat{\pi}_1 \wedge \hat{\pi}_2 \wedge \mathcal{Wfc}(\hat{H}_1 \cup \hat{H}_2))$$

where $\hat{H}_1 \cup \hat{H}_2$ denotes the pairwise union of the components of the symbolic resource and $\mathcal{Wfc}(\hat{H}_1 \cup \hat{H}_2)$ ensures that the composition is well-formed.

5.2 Axiomatic Interface for Consume and Produce

We present our axiomatic interface for the consume and produce operations, used to update the symbolic state during function call and to fold and unfold the predicates. Given the substitution $\hat{\theta} : \text{LVar} \rightarrow_{\text{fin}} \text{SVal}$, the consume and produce operations have the signatures:

$$\text{consume}(m, P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow (\hat{\theta}', \hat{\sigma}_f) \mid \text{abort}(\hat{v}) \quad \text{produce}(P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow \hat{\sigma}'$$

We assume the following holds initially: state $\hat{\sigma}$ and substitution $\hat{\theta}$ are well-formed; and $\hat{\theta}$ covers P for produce, that is, $\text{lv}(P) \subseteq \text{dom}(\hat{\theta})$. For properties 1–4 below, consider the following executions:

$$\begin{array}{ll} \text{consume}(m, P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow (\hat{\theta}', \hat{\sigma}_f) & \text{where } \hat{\sigma} = (\hat{s}, \hat{H}, \hat{\pi}) \text{ and } \hat{\sigma}_f = (\hat{s}', \hat{H}_f, \hat{\pi}') \\ \text{produce}(P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow \hat{\sigma}' & \text{where } \hat{\sigma} = (\hat{s}, \hat{H}, \hat{\pi}) \text{ and } \hat{\sigma}' = (\hat{s}', \hat{H}', \hat{\pi}') \end{array}$$

► **Property 1** (Well-formedness). *The variable store is not altered: that is, $\hat{s}' = \hat{s}$ and*

$$(\text{consume}) \quad \mathcal{Wf}(\hat{\sigma}_f) \text{ and } \mathcal{Wf}(\hat{\theta}', \hat{\pi}') \quad (\text{produce}) \quad \mathcal{Wf}(\hat{\sigma}')$$

► **Property 2** (Path Strengthening). $\hat{\pi}' \Rightarrow \hat{\pi}$

► **Property 3** (Consume Covers P). $\hat{\theta}' \geq \hat{\theta}$ and $\text{dom}(\hat{\theta}') \supseteq \text{lv}(P)$

► **Property 4** (Soundness).

$$(\text{consume}) \quad \exists \hat{H}_P. \quad \hat{H} = \hat{H}_f \cup \hat{H}_P \wedge (\forall \varepsilon, \sigma. \varepsilon, \sigma \models \hat{\sigma}_P \implies \varepsilon(\hat{\theta}'), \sigma \models P) \quad \text{where } \hat{\sigma}_P \triangleq (\emptyset, \hat{H}_P, \hat{\pi}')^a$$

$$(\text{produce}) \quad \exists \hat{H}_P. \quad \hat{H}' = \hat{H} \cup \hat{H}_P \wedge (\forall \varepsilon, \sigma. \varepsilon, \sigma \models \hat{\sigma}_P \implies \varepsilon(\hat{\theta}), \sigma \models P) \quad \text{where } \hat{\sigma}_P \triangleq (\emptyset, \hat{H}_P, \hat{\pi}')$$

► **Property 5** (Completeness: OX consume). *If $\text{abort} \notin \text{consume(OX, } P, \hat{\theta}, \hat{\sigma})$ and $\varepsilon, \sigma \models \hat{\sigma}$, then*

$$\exists \hat{\theta}', \sigma_f, \hat{\sigma}_f. \quad \text{consume(OX, } P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow (\hat{\theta}', \hat{\sigma}_f) \wedge \varepsilon, \sigma_f \models \hat{\sigma}_f$$

► **Property 6** (Completeness: UX consume). *If $\text{consume(UX, } P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow (\hat{\theta}', \hat{\sigma}_f)$, $\varepsilon(\hat{\pi}') = \text{true}$ and $\varepsilon(\hat{\theta}'), (\emptyset, h_P) \models P$, then*

$$\varepsilon, (\emptyset, h_P) \models \hat{\sigma}_P \wedge (\forall h_f. \varepsilon, (s, h_f) \models \hat{\sigma}_f \wedge h_P \# h_f \implies \varepsilon, (s, h_f \uplus h_P) \models \hat{\sigma})$$

► **Property 7** (Completeness: produce). *If $\varepsilon, (s, h) \models \hat{\sigma}$ and $\varepsilon(\hat{\theta}), (\emptyset, h_P) \models P$ and $h \# h_P$, then*

$$\exists \hat{\sigma}_P. \quad \text{produce}(P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow \hat{\sigma} \cdot \hat{\sigma}_P \wedge \varepsilon, (\emptyset, h_P) \models \hat{\sigma}_P$$

^a We choose the empty symbolic store for $\hat{\sigma}_P$; P does not have program variables so this choice is arbitrary. The symbolic state $\hat{\sigma}_P$ is the one used in Prop. 6.

■ **Figure 4** The axiomatic interface for the consume and produce operations.

Recall the use of the consume and produce operations in the function call illustrated in Fig. 2 of §2.2. For $\text{consume}(m, P, \hat{\theta}, \hat{\sigma})$, the initial substitution $\hat{\theta}$ comes from replacing the function parameters with symbolic values given by the arguments in the function call; consume matches the precondition P and substitution $\hat{\theta}$ against part of $\hat{\sigma}$, removing the appropriate resource $\hat{\sigma}_P$ and returning the frame $\hat{\sigma}_f$ and the substitution $\hat{\theta}'$ which extends $\hat{\theta}$ with further information given by the match. For $\text{produce}(Q_{ok}, \hat{\theta}', \hat{\sigma}_f)$, the produce takes the postcondition Q_{ok} and this resulting substitution $\hat{\theta}'$ and creates a symbolic state which is composed with $\hat{\sigma}_f$ to obtain $\hat{\sigma}'$. Notice that the consume operation can abort with error information if no match is found. The produce operation does not abort, but it may render states unsatisfiable, in which case the branch is cut.

In Fig. 4, we present the axiomatic interface of the consume and produce operations, identifying sufficient properties to prove OX and UX soundness for the function-call rule and the folding and unfolding of predicates, as we demonstrate in the next section (§5.3). Properties 1–3 ensure that the operations are compatible with the expected properties of symbolic execution, including well-formedness $\mathcal{Wf}(\hat{\theta}', \hat{\pi}')$ of the symbolic substitutions with respect to path conditions. This property guarantees that the $\hat{\pi}'$ implies that $\hat{\theta}'$ does not map logical variables into \perp , that is, $\hat{\pi}' \models \text{codom}(\hat{\theta}') \subseteq \text{Val}$.

Properties 4–7 give conditions for `consume` and `produce` to soundly decompose and compose symbolic states respectively, while being compatible with symbolic and logical interpretations. These properties will come as no surprise to those with a formal knowledge of symbolic execution. However, their identification was not easy, requiring a considerable amount of back and forth between the soundness proof and the properties to pin them down properly. We describe the more interesting of the properties described in Fig. 4:

- Prop. 2 states that path conditions may only get strengthened: OX and UX `consume` may strengthen $\hat{\pi}$ to due to branching; additionally UX `consume` may strengthen $\hat{\pi}$ arbitrarily due to cutting; and `produce` may add constraints to $\hat{\pi}$ arising from P .
- Prop. 4, for `consume` (and similarly for `produce`), states that the operation is sound: the symbolic resource of $\hat{\sigma}$ can be decomposed as $\hat{H} = \hat{H}_f \cup \hat{H}_P$, i.e., it consists of the symbolic resources of $\hat{\sigma}_P$ and $\hat{\sigma}_f$, respectively, and $\forall \varepsilon, \sigma. \varepsilon, \sigma \models \hat{\sigma}_P \implies \varepsilon(\hat{\theta}''), \sigma \models P$ states that all models of $\hat{\sigma}_P$ are models of P .
- Prop. 5 captures that successful OX consumptions do not drop paths: if no branch aborts, and we have a model $\varepsilon, \sigma \models \hat{\sigma}$, then there exists a branch $\hat{\sigma}_f$ with a model using the same ε , i.e., there exist σ_f such that $\varepsilon, \sigma_f \models \hat{\sigma}_f$.
- Prop. 6 is as follows: in successful UX `consume`, any model of the consumed assertion P must also model the consumed state $\hat{\sigma}_P$ (obtained from Prop. 4), and when extended with a compatible model of the output state $\hat{\sigma}_f$ it must model the input state $\hat{\sigma}$.

Assuming these properties of the `consume` and `produce` operations, we are able to prove that the function-call rule and the predicate folding and unfolding are sound, and thus that our whole CSE engine is sound (Thm. 3). In §5.3, we give an example (Ex. 2) illustrating some of the properties in action during a function call.

5.3 Full CSE Engine

We introduce our full CSE engine, extending the core CSE engine (§4) with the ability to soundly call valid SL and ISL function specifications (§5.1), and to fold/unfold predicates. We extend and adapt the compositional symbolic operational semantics to carry a specification context Γ and the mode of execution m (OX or UX), obtaining the judgement $\hat{\sigma}, \mathsf{C} \Downarrow_{\Gamma}^m o : \hat{\sigma}'$, and extend the possible outcomes with *abort*, as `consume` can abort. The rules are analogous except for the rules for function calls and predicate folding/unfolding, as detailed below.⁴

Function-Call Rule. The unified success rule for a function call is in Fig. 5, using the notation $\Gamma(f)|_m$ to isolate the m -mode specifications of f . A description of each step is included in the rule itself. In short, given an initial state $\hat{\sigma}$, the rule selects a function specification, consumes the specification pre-condition from $\hat{\sigma}$, resulting in $\hat{\sigma}'$, and produces the post-condition of the specification into $\hat{\sigma}'$, resulting in the final state $\hat{\sigma}''$. The steps in grey are uninteresting (about renamings and fresh variables) and can be ignored on a first reading.

► **Example 2.** We show a possible execution of a function call using a function specification, where we assume we have been given `consume` and `produce` example implementations that satisfy the axiomatic interface. Consider the function f given in §2.1 and the ISL specification given in §2.3: $[c = c * x = x * P] f(c, x) [ok : x \mapsto c * c \geq 42 * \text{ret} = v]$ where P is $x \mapsto v$,

⁴ The satisfiability check $\mathbf{SAT}(\hat{\pi})$ used by the rules over-approximates the existence of a model for $(\hat{s}, \hat{H}, \hat{\pi})$, due to the presence of symbolic predicates; for sound reasoning in UX mode, our engine addresses this source of over-approximation by under-approximating the satisfiability check once by bounded unfolding of predicates at the end of execution.

(1) $\langle\!\langle \vec{x} = \vec{x} * P \rangle\!\rangle f(\vec{x}) \langle\!\langle ok : Q_{ok} \rangle\!\rangle \langle\!\langle err : Q_{err} \rangle\!\rangle \in \Gamma(f) _m$	get function specification
(2) $\llbracket \vec{E} \rrbracket_{\hat{s}}^{\hat{\pi}} \Downarrow (\vec{\theta}, \hat{\pi}')$ and $\hat{\theta} \triangleq \{\vec{v}/\vec{x}\}$	evaluate function parameters
(3) $\text{consume}(m, P, \hat{\theta}, \hat{\sigma}[\text{pc} := \hat{\pi}']) \rightsquigarrow (\hat{\theta}', \hat{\sigma}')$	consume pre-condition as Q_{ok} is a post-condition
(4) $Q_{ok} = \exists \vec{y}. Q'_{ok}$	extend substitution to cover Q_{ok}
(5) $\hat{\theta}'' \triangleq \hat{\theta}' \cup \{\vec{z}/\vec{y}\}$	fresh variables
(6) \vec{z}, r, \hat{r} fresh	
(7) $Q''_{ok} = Q'_{ok}\{r/\text{ret}\}$ and $\hat{\theta}''' = \hat{\theta}'' \cup \{\hat{r}/r\}$	set up return value
(8) $\text{produce}(Q''_{ok}, \hat{\theta}''', \hat{\sigma}') \rightsquigarrow \hat{\sigma}''$	produce post-condition
$\hat{\sigma}, y := f(\vec{E}) \Downarrow_{\Gamma}^m ok : \hat{\sigma}''[\text{sto} := \hat{s}[y \mapsto \hat{r}]]$	

Figure 5 Unified function-call rule for CSE: success case, where $\hat{\sigma} = (\hat{s}, \hat{H}, \hat{\pi})$.

here assumed to be in the function specification context Γ . Suppose the symbolic execution is in a state $\hat{\sigma} = (\hat{s}, \hat{H}, \hat{\pi})$ and that the next step is $\hat{\sigma}, y := f(50, 1) \Downarrow_{\Gamma}^{\text{UX}} ok : \hat{\sigma}'$. Let $\hat{H} = (\{\hat{x} \mapsto \hat{c}, \hat{y} \mapsto 1, 3 \mapsto 5\}, \emptyset)$ be the symbolic resource, and $\hat{\pi} = \hat{c} \geq 42 \wedge \hat{x} \neq \hat{y} \wedge \hat{x}, \hat{y} \in \text{Nat}$ be the symbolic path condition (the symbolic store \hat{s} is irrelevant to this computation and left opaque). We now follow the steps (1) - (8) described in the function-call rule in Fig. 5.

Step (1) is above. Step (2) evaluates the parameters of the function call which in this case yields the initial substitution is $\hat{\theta} = \{50/c, 1/x\}$. Step (3) is to consume the pre-condition of f : $\hat{\theta}$ identifies the logical variable x with 1, and thus, this $\hat{\theta}$ maps P into $1 \mapsto v$; now we check whether there exists a resource in \hat{H} that matches this. There are two possibilities: either $\hat{x} = 1$ and $v = \hat{c}$; or $\hat{y} = 1$ and $v = 1$. Let us choose the first match. Thus, with our axiomatic description of a consume operation, $\text{consume}(\text{UX}, x \mapsto v, \hat{\theta}, \hat{\sigma})$ gives the pair $(\hat{\theta}', \hat{\sigma}_f)$, with substitution $\hat{\theta}' = \{50/c, 1/x, \hat{c}/v\}$ and symbolic frame $\hat{\sigma}_f = (\hat{s}, (\{\hat{y} \mapsto 1, 3 \mapsto 5\}, \emptyset), \hat{\pi} \wedge \hat{x} = 1)$. Here $\hat{\theta}' \geq \hat{\theta}$ as described by Prop. 3 of Fig. 4, the new path condition $\hat{\pi} \wedge \hat{x} = 1$ is stronger than the initial $\hat{\pi}$, as required by Prop. 2.

Steps (4) - (5) are straightforward: Q_{ok} is not existentially quantified and the domain of $\hat{\theta}'$ covers $Q_{ok} = x \mapsto c * c \geq 42 * \text{ret} = v$. Steps (6) - (7) set up the return value by renaming ret with a fresh logical variable r as in $Q''_{ok} = Q_{ok}\{r/\text{ret}\}$ and defining the substitution $\hat{\theta}'' = \hat{\theta}' \uplus \{\hat{r}/r\}$, with \hat{r} a fresh symbolic variable. Step (8) produces the post-condition which results in $(\hat{s}, (\{\hat{y} \mapsto 1, 3 \mapsto 5, 1 \mapsto 50\}, \emptyset), \hat{\pi}')$, for some $\hat{\pi}'$ that is satisfiable.

We illustrate the general execution of the function-call rule in Fig. 6. Successful `consume` may branch (in the figure: $\hat{\sigma}_{f_1}, \dots, \hat{\sigma}_{f_k}$) due to different ways of matching with the symbolic state $\hat{\sigma}$, and the function call will branch accordingly. In both modes, in each successful branch, say with frame state $\hat{\sigma}_{f_i}$, the function-call rule will call `produce`, which will produce both Q_{ok} and Q_{err} postconditions of the function specification. The function call propagates errors from `consume`, whose error handling can depend on the mode of reasoning. In OX mode, all errors must be reported; the figure shows an example with two *abort* outcomes, one with a symbolic value $\hat{\sigma}_{miss}$, representing a missing outcome, and another abort $\hat{\sigma}_{\hat{v}}$. In UX mode, in contrast, errors can be cut: e.g., a `consume` implementation may choose to report missing errors (to be used in e.g. bi-abduction, see §7), but cut other errors, as illustrated in the figure. Lastly, note that `consume` implementations must represent missing-resource errors as abort errors. To see why, consider the function `do_nothing() { skip; return null }` and the (nonsensical but valid) specification `[5 ↦ 0] do_nothing() [ok : 5 ↦ 0 * ret = null]`. Of course, in the concrete semantics, calling the function will never result in a miss. Now, say the symbolic engine calls the function using the provided function specification. If the the resource of the pre-condition is not available in the current symbolic heap, then the consumption of the pre-condition will fail. Because no concrete execution of the function results in a miss, it would be unsound for the consumption to report a missing-resource error in this case.

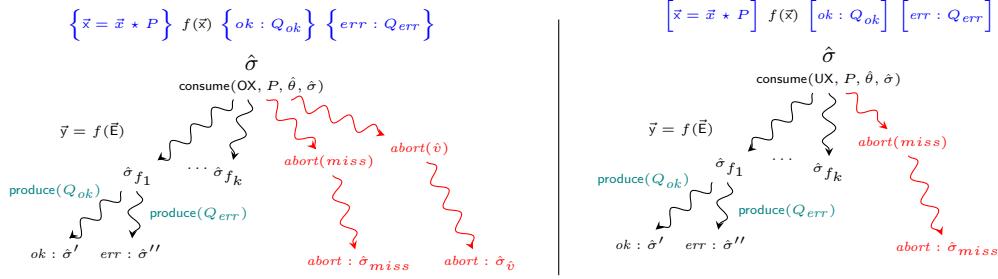


Figure 6 Branching in OX and UX function calls.

Predicate Rules. To handle the folding and unfolding of predicates in symbolic states, we extend the language syntax with the following two ghost commands (also known as tactic commands): $C \in \text{Cmd} ::= \dots \mid \text{fold } p(\vec{E}) \mid \text{unfold } p(\vec{E})$, where $\vec{E} \in \text{PExp}$ specifies the values of the in-parameters of the predicate p . In the concrete semantics, these commands are no-ops, as they are ghost commands. The symbolic-semantics rules are similar to the function-call rule: in short, a fold of a predicate consumes the body of the predicate, learns the out-parameters of the predicate, and adds the predicate (with the specified in-parameters and learnt out-parameters) to the symbolic state; and an unfold of a predicate finds a corresponding predicate in the symbolic state, learns the out-parameters of the predicate, and produces the body of the predicate.

Soundness. Our CSE engine is sound: OX soundness, expectedly, has no restrictions on the predicates; UX soundness, on the other hand, allows only strictly exact predicates (i.e., predicates whose bodies are satisfiable by at most one heap [33]) to be folded to ensure that no information is dropped.

► **Theorem 3** (Compositional OX and UX soundness). *If all UX predicate foldings are limited to strictly exact predicates, then the following hold:*

$$\begin{aligned} &\models (\gamma, \Gamma) \wedge \hat{\sigma}, C \Downarrow_{\Gamma}^{OX} \hat{\Sigma}' \wedge \text{abort} \notin \hat{\Sigma}' \wedge \varepsilon, \sigma \models \hat{\sigma} \wedge \sigma, C \Downarrow_{\gamma} o : \sigma' \implies \\ &\quad \exists \hat{\sigma}', \varepsilon' \geq \varepsilon. (o, \hat{\sigma}') \in \hat{\Sigma}' \wedge \varepsilon', \sigma' \models \hat{\sigma}' \\ &\models (\gamma, \Gamma) \wedge \hat{\sigma}, C \Downarrow_{\Gamma}^{UX} o : \hat{\sigma}' \wedge \varepsilon, \sigma' \models \hat{\sigma}' \implies \exists \sigma. \varepsilon, \sigma \models \hat{\sigma} \wedge \sigma, C \Downarrow_{\gamma} o : \sigma' \end{aligned}$$

Proof. Proofs of rules not related to function calls and predicates are the same as for Thm. 1. OX soundness of function call follows from soundness of `consume` (Prop. 4), OX completeness of `consume` (Prop. 5), and completeness of `produce` (Prop. 7). UX soundness of function call follows from the soundness of `consume` and `produce` (Prop. 4) and UX completeness of `consume` (Prop. 6). Full details are given in the extended version of this paper [18]. ◀

6 Consume and Produce Implementations

We provide implementations for the `consume` and `produce` operations and prove that they satisfy the properties 1–7 of the axiomatic interface (§5.2). We give the complete set of rules implementing these operations in the extended version [18] and only discuss the interesting rules here. Our implementations are inspired by the Gillian OX implementations, although previous work has only given a brief informal sketch of these implementations [10].

6.1 Implementations

Consume Implementation. As is typical for SL-based analysis tools, our *consume* operation works with a fragment of the assertions with no implications, disjunctions, or existentials (which are handled outside consumption, see, e.g., the function-call rule); which means that input assertions for consumption are \star -separated lists of simple assertions. Following the implementation of Gillian, our *consume* operation works by consuming one simple assertion at a time and is split into two phases, a planning phase and a consumption phase:

$$\text{consume}(m, P, \hat{\theta}, \hat{\sigma}) \triangleq \text{let } mp = \text{plan}(\text{dom}(\hat{\theta}), P) \text{ in } \text{consume}_{\text{MP}}(m, mp, \hat{\theta}, \hat{\sigma})$$

Here our interest lies in the consumption phase: the planning phase of Gillian has been formalised and discussed by Lööw et al. [19]. We, however, repeat the necessary background of the planning phase here to keep this paper self-contained.

Consumption Planning. The *plan* operation has two responsibilities: to resolve the order of consumption and the unknown variables. The operation takes a set of known logical variables (above, $\text{dom}(\hat{\theta})$) and an assertion P to plan and returns a *matching plan* (*MP*) of the form $[(\text{Asrt}, [(\text{LVar}, \text{LExp})])]$. An MP for an assertion $P = P_1 \star \dots \star P_n$ ensures that (1) the simple assertions P_i of P are consumed in an order such that the *in-parameters* (*ins*) of each simple assertion, i.e., the parameters (logical variables) that must be known to consume the simple assertion, have been learnt during previous consumption; (2) specifies how *out-parameters* (*outs*) are learnt during consumption, i.e., the remaining parameters (logical variables). For instance, the in-parameter of the cell-assertion $x \mapsto z + 1$ is x and the out-parameter is z , where the value of z can be learnt by inspecting the heap and subtracting 1. Another example is given by the pure assertion $x + 1 = y + 3$: here, what the in- and out-parameters are depend on what variables are known, e.g., if we know x we can learn y and vice versa.

► **Example 4.** Say we are to plan the assertion $x \leq 10 \star x \mapsto y \star y = z - 10$ knowing that $\hat{\theta} = \{\hat{x}/x\}$, that is, x is known but y and z are not. One MP for this assertion is $[(x \leq 10, []), (x \mapsto y, [(y, O)]), (y = z - 10, [(z, y + 10)])]$, where O is used to refer to the value of the consumed heap cell. First, by consuming $x \leq 10$ we learn nothing (x is already known); second, when consuming $x \mapsto y$ we learn y (from the consumed heap cell); third, since we learn y in the previous step, we can learn z by manipulating the assertion to $z = y + 10$. Another MP is $[(x \mapsto y, [(y, O)]), (x \leq 10, []), (y = z - 10, [(z, y + 10)])]$. Note that there is no MP starting with assertion $y = z - 10$, because y and z are not known initially.

Consuming Assertions. Having discussed the planning phase, we now discuss how pure assertions, cell assertions, and predicate assertions are consumed.

Consuming Pure Assertions. Fig. 7 contains the $\text{consume}_{\text{MP}}$ rules for consuming pure assertions. The rules are defined in terms of the helper operation $\text{consPure}(m, \hat{\pi}, \hat{\pi}') = \hat{\pi}'' \mid \text{abort}$ which depends on the current reasoning mode m :

- (i) for $m = \text{OX}$, we check $\neg \text{SAT}(\hat{\pi} \wedge \neg \hat{\pi}')$ which is equivalent to $\hat{\pi} \Rightarrow \hat{\pi}'$, hence, the SAT check corresponds to the entailment check seen in traditional OX reasoning;
- (ii) for $m = \text{OX}$, if $\text{SAT}(\hat{\pi} \wedge \neg \hat{\pi}')$, that is, $\neg(\hat{\pi} \Rightarrow \hat{\pi}')$, consumption, of course, must abort;
- (iii) for $m = \text{UX}$, consPure instead cuts all paths of $\hat{\pi}$ that are not compatible with the input pure assertion $\hat{\pi}'$, i.e., forms $\hat{\pi} \wedge \hat{\pi}'$, and then checks if there are any paths left after the cut, i.e., checks $\text{SAT}(\hat{\pi} \wedge \hat{\pi}')$.

$$\begin{aligned}
 \text{consPure}(m, \hat{\pi}, \hat{\pi}') = & \\
 \begin{cases} \hat{\pi} \wedge \hat{\pi}', & \text{if } m = \text{UX and } \text{SAT}(\hat{\pi} \wedge \hat{\pi}') \\ \hat{\pi}, & \text{if } m = \text{OX and } \neg \text{SAT}(\hat{\pi} \wedge \neg \hat{\pi}') \\ \text{abort}, & \text{if } m = \text{OX and } \text{SAT}(\hat{\pi} \wedge \neg \hat{\pi}') \end{cases} &
 \frac{P \text{ is pure} \quad \text{outs} = [(x_i, E_i)]_{i=1}^n \quad \hat{\theta}' = \hat{\theta} \uplus \{(\hat{\theta}(E_i)/x_i)\}_{i=1}^n \quad \text{consPure}(m, \hat{\pi}, \hat{\theta}'(P)) = \hat{\pi}'}{\text{consume}_{\text{MP}}(m, [(P, \text{outs})], \hat{\theta}, \hat{\sigma}) \rightsquigarrow (\hat{\theta}', \hat{\sigma}[\text{pc} := \hat{\pi}']))} \\
 & \\
 \frac{P \text{ is pure} \quad \text{outs} = [(x_i, E_i)]_{i=1}^n \quad \hat{\theta}' = \hat{\theta} \uplus \{(\hat{\theta}(E_i)/x_i)\}_{i=1}^n \quad \text{consPure(OX, } \hat{\pi}, \hat{\theta}'(P)) = \text{abort}}{\text{consume}_{\text{MP}}(\text{OX}, [(P, \text{outs})], \hat{\theta}, \hat{\sigma}) \rightsquigarrow \text{abort}([\text{"consPure"}, \hat{\theta}'(P), \hat{\pi}])}
 \end{aligned}$$

■ **Figure 7** Rules for `consPure` and `consumeMP` (excerpt), where $\hat{\sigma} = (\hat{s}, \hat{H}, \hat{\pi})$.

$$\begin{aligned}
 \frac{\hat{h} = \hat{h}_f \uplus \{\hat{v}_1 \mapsto \hat{v}_2\} \quad \hat{\pi}' = \hat{\pi} \wedge (\hat{v} = \hat{v}_1) \quad \text{SAT}(\hat{\pi}')}{\text{consCell}(\hat{v}, \hat{\sigma}) \rightsquigarrow (\hat{v}_2, \hat{\sigma}[\text{heap} := \hat{h}_f, \text{pc} := \hat{\pi}'])} & \quad \frac{\text{SAT}(\hat{\pi} \wedge \hat{v} \notin \text{dom}(\hat{h}))}{\text{consCell}(\hat{v}, \hat{\sigma}) \rightsquigarrow \text{abort}} \\
 & \\
 \frac{\text{consPure}(m, \hat{\pi}, \hat{\theta}(E_a) \in \text{Nat}) = \hat{\pi}' \quad \text{consCell}(\hat{\theta}(E_a), \hat{\sigma}[\text{pc} := \hat{\pi}']) \rightsquigarrow (\hat{v}, \hat{\sigma}') \quad \hat{\theta}_{\text{subst}} = \{\hat{v}/\text{O}\} \quad \text{outs} = [(x_i, E_i)]_{i=1}^n \text{ and } ((\hat{\theta} \uplus \hat{\theta}_{\text{subst}})(E_i) = \hat{v}_i)_{i=1}^n \quad \hat{\theta}' = \hat{\theta} \uplus \{(\hat{v}_i/x_i)\}_{i=1}^n \quad \text{consPure}(m, (\hat{\sigma}').\text{pc}, \hat{\theta}'(E_v) = \hat{v}) = \hat{\pi}''}{\text{check for evaluation error} \quad \text{branch over all cell consumptions} \quad \text{substitution with cell contents} \quad \text{collect and instantiate outs} \quad \text{extend substitution with outs} \quad \text{consume cell contents}} \\
 & \\
 & \text{consume}_{\text{MP}}(m, [(E_a \mapsto E_v, \text{outs})], \hat{\theta}, \hat{\sigma}) \rightsquigarrow (\hat{\theta}', \hat{\sigma}'[\text{pc} := \hat{\pi}''])
 \end{aligned}$$

■ **Figure 8** Rules for `consCell` and `consumeMP` (excerpt), where $\hat{\sigma} = (\hat{s}, \hat{H}, \hat{\pi})$.

► **Example 5.** To exemplify the difference between OX and UX consume, consider calling a function `foo(y)` with the precondition $y = y \star y \geq 0$. The first step of calling a function using its function specification is to consume its precondition, which we now illustrate. Say we are in a symbolic state with path condition $\hat{\pi} = \hat{v} > 5$ and are calling the function with an argument that symbolically evaluates to \hat{v} , i.e., we know $\hat{\theta}(y) = \hat{v}$. In OX mode, the function call aborts: `consumeMP`'s pure consumption error rule is applicable because $\text{consPure(OX, } \hat{\pi}, \hat{\theta}(y) \geq 10) = \text{abort}$ since $\text{SAT}(\hat{\pi} \wedge \neg(\hat{v} \geq 10))$. Intuitively, this means that not all paths described by $\hat{\pi}$ are described by $y \geq 10$, i.e., we are “outside” the precondition of the function. Differently, in UX mode, a call to `consPure(UX, } \hat{\pi}, \hat{\theta}(y) \geq 10)` cuts the incompatible paths by strengthen the path condition to $\hat{\pi} \wedge \hat{v} \geq 10$. That is, instead of as in OX mode where execution must abort, in UX mode the execution can continue.

Consuming Cell Assertions. Fig. 8 contains some of the `consumeMP` rules for consuming a cell assertion. The rules are defined using the helper operation $\text{consCell}(\hat{v}, \hat{\sigma}) \rightsquigarrow (\hat{v}', \hat{\sigma}') \mid \text{abort}$, which tries to consume the cell at address \hat{v} in mode m by branching over all possible addresses in the heap, returning the corresponding value in the heap, \hat{v}' , and the rest of the state, $\hat{\sigma}'$, and returns *abort* if it is possible for the address \hat{v} to point outside of heap. In the successful `consumeMP` rule (featured in Fig. 8), the operation `consPure` is used to consume the contents of the matched cells. The erroneous `consumeMP` rules are available in [18].

Consuming Predicate Assertions. The `consumeMP` rules for predicate assertions are generalisations of the rules for cell assertions, with two main differences: predicates may have multiple *ins* and *outs* whereas cells have a single *in* and single *out*, and predicate assertions refers to symbolic predicates whereas cell assertions refer to the symbolic heap.

$$\begin{array}{c}
 \frac{\hat{h}' \triangleq \hat{h} \uplus \{\hat{l} \mapsto \hat{v}_\emptyset\} \quad \hat{\pi}' \triangleq \hat{\pi} \wedge \hat{l} \notin \text{dom}(\hat{h}) \quad \text{SAT}(\hat{\pi}') \quad \hat{\sigma}' \triangleq \hat{\sigma}[\text{heap} := \hat{h}', \text{pc} := \hat{\pi}']}{\text{prodCell}(\hat{l}, \hat{v}_\emptyset, \hat{\sigma}) = \hat{\sigma}'} \quad \frac{P \text{ pure} \quad \hat{\pi}' \triangleq \hat{\pi} \wedge \hat{\theta}(P) \quad \text{SAT}(\hat{\pi}') \quad \hat{\sigma}' \triangleq \hat{\sigma}[\text{pc} := \hat{\pi}']}{\text{produce}(P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow \hat{\sigma}'} \quad \frac{\hat{\pi}' \triangleq \hat{\pi} \wedge \hat{\theta}(E_a) \in \text{Nat} \wedge \hat{\theta}(E_v) \in \text{Val} \quad \text{prodCell}(\hat{\theta}(E_a), \hat{\theta}(E_v), \hat{\sigma}[\text{pc} := \hat{\pi}']) = \hat{\sigma}'}{\text{produce}(E_a \mapsto E_v, \hat{\theta}, \hat{\sigma}) \rightsquigarrow \hat{\sigma}'}
 \end{array}$$

Figure 9 Rules for `prodCell` and `produce` (excerpt), where $\hat{\sigma} = (\hat{s}, \hat{H}, \hat{\pi})$ and \hat{v}_\emptyset denotes a symbolic value or \emptyset .

Produce Implementation. The implementation of $\text{produce}(Q, \hat{\theta}, \hat{\sigma})$ is straightforward: it extends $\hat{\sigma}$ with the symbolic state corresponding to Q given $\hat{\theta}$, ensuring well-formedness. Unlike `consume`, `produce` does not require planning and is not dependent on the mode of execution. An excerpt of rules for `produce` is given in Fig. 9. Like `consume`, `produce` does not support assertion-level implications, which are usually not found in function specifications or predicate definitions. However, `produce`, unlike `consume`, supports assertion-level disjunctions since the function specification we synthesise using bi-abduction contains disjunctions (cf. §8).

6.2 Correctness of Implementations

The correctness of the `consume` and `produce` implementations amount to showing that they satisfy properties of the axiomatic interface for `consume` and `produce`.

► **Theorem 6** (Correctness). *The `consume` and `produce` operations satisfy properties 1-7 (§5.2).*

7 Bi-abduction

To enable hosting Pulse-style true bug-finding on top of our CSE engine, it must support UX bi-abduction. In this section, we show how the engine presented in §5 can be extended to support UX bi-abduction by catching missing-resource errors that happen during execution and applying *fixes* to enable uninterrupted execution instead of faulting. These fixes, stored in an *anti-frame*, add the missing resource to the current state and allow execution to continue. This style of bi-abduction was introduced in the OX setting by JaVert 2.0 [10]. Here we show that it can also be applied to the UX setting. We focus on UX bi-abduction for true bug-finding, but also discuss in §8 how the obtained UX results can be used in an OX setting.

We introduce the judgement for the bi-abductive symbolic engine, $\hat{\sigma}, C \Downarrow_{\Gamma}^{\text{bi}} o : (\hat{\sigma}', \hat{H})$, with outcomes $o ::= ok \mid err$, and the anti-frame $\hat{H} = (\hat{h}, \hat{P})$ containing the anti-heap \hat{h} and the anti-predicates \hat{P} . We do not need *miss* or *abort* as possible outcomes since if they happen during execution they will be either fixed by bi-abduction or cut if not. The new judgement is defined in terms of the judgement $\hat{\sigma}, C \Downarrow_{\Gamma}^{\text{UX}} o : \hat{\sigma}'$ and a partial function `fix` as:

$$\begin{array}{c}
 \text{BIAB} \\
 \frac{\hat{\sigma}, C \Downarrow_{\Gamma}^{\text{UX}} o : \hat{\sigma}' \quad \text{not_Seq}(C) \quad o \notin \{miss, abort\}}{\hat{\sigma}, C \Downarrow_{\Gamma}^{\text{bi}} o : (\hat{\sigma}', (\emptyset, \emptyset))} \quad \text{BIAB-MISS} \\
 \frac{\hat{\sigma}, C \Downarrow_{\Gamma}^{\text{UX}} o : \hat{\sigma}' \quad \text{not_Seq}(C) \quad o \in \{miss, abort\} \quad \text{fix}(\hat{\sigma}') = (\hat{H}, \hat{\pi}) \quad \hat{\sigma} \cdot (\hat{H}, \hat{\pi}), C \Downarrow_{\Gamma}^{\text{bi}} o' : (\hat{\sigma}'', \hat{H}')} {\hat{\sigma}, C \Downarrow_{\Gamma}^{\text{bi}} o' : (\hat{\sigma}'', \hat{H} \cup \hat{H}')}
 \end{array}$$

$$\begin{array}{c}
 \text{BIAB-SEQ-ERR} \\
 \frac{\hat{\sigma}, C_1 \Downarrow_{\Gamma}^{\text{bi}} o : (\hat{\sigma}', \hat{H}) \quad o \neq ok}{\hat{\sigma}, C_1; C_2 \Downarrow_{\Gamma}^{\text{bi}} o : (\hat{\sigma}', \hat{H})}
 \end{array}
 \quad
 \begin{array}{c}
 \text{BIAB-SEQ} \\
 \frac{\hat{\sigma}, C_1 \Downarrow_{\Gamma}^{\text{bi}} ok : (\hat{\sigma}', \hat{H}_1) \quad \hat{\sigma}', C_2 \Downarrow_{\Gamma}^{\text{bi}} o : (\hat{\sigma}'', \hat{H}_2) \quad \text{sv}(\text{dom}(\hat{H}_2)) \cap (\text{sv}(\hat{\sigma}') \setminus \text{sv}(\hat{\sigma})) = \emptyset}{\hat{\sigma}, C_1; C_2 \Downarrow_{\Gamma}^{\text{bi}} o : (\hat{\sigma}'', \hat{H}_1 \cup \hat{H}_2)}
 \end{array}$$

where `not_Seq(C)` denotes that C is not a sequence command (i.e., does not have the form $C_1; C_2$), $\hat{\sigma} \cdot (\hat{H}, \hat{\pi})$ denotes $\hat{\sigma} \cdot (\emptyset, \hat{H}, \hat{\pi} \wedge \mathcal{Wfc}(\hat{H}))$, $\text{dom}(\hat{H}_2) = \text{sv}(\text{dom}(\hat{h}_2)) \cup \text{sv}(\text{dom}(\hat{P}_2))$, and $\text{dom}(\hat{P})$ denotes all symbolic variables of the `ins` of the predicates in \hat{P} . The rule BIAB states that for non-erroneous outcomes, the bi-abductive engine has the same semantics as the underlying UX engine it is built on top of. The rule BIAB-MISS, which is the most interesting rule, catches missing-resource errors from the underlying UX engine and uses the `fix` function to add the missing resource to the current symbolic state and anti-frame, such that execution can continue. The two rules BIAB-SEQ-ERR and BIAB-SEQ are two straightforward sequencing rules for the engine, where the symbolic-variable condition of BIAB-SEQ ensures that the anti-frame \hat{H}_2 does not clash with resource allocated by C_1 .

To exemplify, say the engine is in symbolic state $(\{v \mapsto 0, a \mapsto 13\}, (\emptyset, \emptyset), \text{true})$ and is about execute $v := [a]$, i.e., about to retrieve the value of the heap cell with address a . Since this cell is not in the heap, the rule LOOKUP-ERR-MISSING from Fig. 3 is applicable, which sets the variable `err` to `[“MissingCell”, “a”, 13]` and gives outcome `miss`. Now, in the rule BIAB-MISS, given the data in the `err` variable, the `fix` function constructs a fix $((\{13 \mapsto \hat{v}\}, \emptyset), \text{true})$ where \hat{v} is a fresh variable. The rule adds this fix to both the current symbolic state and the anti-frame, resulting in the symbolic state $(\{v \mapsto \hat{v}, a \mapsto 13\}, (\{13 \mapsto \hat{v}\}, \emptyset), \text{true})$ and outcome `ok`. Other cases are similar. E.g., when abort outcomes from `consume` represent missing resource (e.g., when invoked in a function call), `fix` returns the resources needed for the execution to continue. The following theorem captures the essence of bi-abduction:

► **Theorem 7** (CSE with Bi-Abduction: UX Soundness).

$$\hat{\sigma}, C \Downarrow_{\Gamma}^{bi} o : (\hat{\sigma}', \hat{H}) \implies \hat{\sigma} \cdot (\hat{H}, \text{true}), C \Downarrow_{\Gamma}^{UX} o : \hat{\sigma}'$$

8 Analysis Applications

We discuss the three analysis applications we have built on top of our unified CSE engine, to demonstrate its wide applicability: EX whole-program automatic symbolic testing (§8.1); OX semi-automatic verification (§8.2); and UX automatic true bug-finding (§8.3).

We have gathered these three analyses from different corners of the literature. EX symbolic testing is well understood in the first-order symbolic execution literature. OX verification is well understood in the consume-produce symbolic execution literature. However, one novelty here is that the correctness proof of the analysis is established with respect to our axiomatic interface rather than the consume/produce operations directly, allowing us to show that function specifications are valid w.r.t. the standard SL definition of validity. In contrast to the other two analyses, UX bug-finding has not previously been implemented in consume-produce style, making this a novel contribution. To simplify the presentation, we consider only non-recursive functions. All applications can be extended to handle bounded recursion by adding a fuel parameter. Unbounded recursion can be handled in verification via user-provided annotations, but is not a good fit for automatic bug-finding.

8.1 EX Whole-program Symbolic Testing

Our EX core engine allows us to implement simple non-compositional analyses, such as whole-program symbolic testing, in the style of CBMC [16] and Gillian [11]. For this analysis, we augment the input language with three additional commands: `x := sym`, for creating symbolic variables; `assume(E)`, for imposing a constraint E on the current state; and `assert(E)`, for checking that E is true in the current state. The operational semantics for these commands is given in the extended version of this paper [18].

The testing algorithm is as follows. Given a command C and implementation context γ , the analysis starts from the state $\hat{\sigma} \triangleq (\{x \mapsto \text{null} \mid x \in \text{pv}(C)\}, \emptyset, \text{true})$, and executes C to completion. The analysis reports back any violations of the `assert` commands encountered during execution. Given the core engine is UX sound, any bug found will be a true bug. Moreover, if the analysed code contains no unbounded recursion, given the core engine is OX sound, all existing bugs will be found modulo the ability of the underlying SMT solver.

8.2 OX Verification

We formalise an OX verification procedure, `verifyOX`, on top of our CSE engine. Given a specification context Γ , a function $f(\vec{x}) \{ C; \text{return } E \}$ with $f \notin \text{dom}(\Gamma)$, and an OX specification $t_f = \{\vec{x} = \vec{x} \star P\} \{ok : Q_{ok}\} \{err : Q_{err}\}$, if $\text{verifyOX}(\Gamma, f, t_f)$ terminates successfully, then we can soundly extend Γ to $\Gamma' = \Gamma[f \mapsto t_f]$. The algorithm is given below:

1. Let $\hat{\theta} \triangleq \{\hat{x}/x \mid x \in \text{lv}(\vec{x} = \vec{x} \star P)\}$, $\hat{s} \triangleq \{x \mapsto \hat{x} \mid x \in \vec{x}\} \cup \{x \mapsto \text{null} \mid x \in \text{pv}(C) \setminus \vec{x}\}$, and $\hat{\sigma} = (\hat{s}, \emptyset, \text{true})$.
2. Set up symbolic state corresponding to pre-condition: $\text{produce}(P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow \hat{\sigma}'$.
3. Execute the function to completion: $\hat{\sigma}', C; \text{ret} := E \Downarrow_{\Gamma}^{\text{OX}} \hat{\Sigma}'$. Then, for every $(o, \hat{\sigma}'') \in \hat{\Sigma}'$:
 - (a) If $o = \text{miss}$ or $o = \text{abort}$, abort with an error.
 - (b) If $o = ok$, then let $\hat{\theta}' = \hat{\theta} \uplus \{(\hat{\sigma}''.\text{sto})(\text{ret})/r\}$ for a fresh r and let $Q' = Q_{ok}\{r/\text{ret}\}$. Otherwise, $o = err$, in which case let $\hat{\theta}' = \hat{\theta}$ and $Q' = Q_{err}$.
 - (c) Consume the post-condition: $\text{consume(OX, } Q'', \hat{\theta}', \hat{\sigma}'') \rightsquigarrow (\hat{\theta}'', \hat{\sigma}''')$, where $Q' = \exists \vec{y}. Q''$.
 - (d) If consumption fails or the final heap is not empty, abort with an error.⁵

8.3 UX Specification Synthesis and True Bug-finding

Recall that Pulse-style UX bug-finding is powered by UX specification synthesis, where, after appropriate filtering, synthesised erroneous specification can be reported as bugs. UX specification synthesis, in turn, is powered by UX bi-abduction (as introduced in §7).

To formalise the specification synthesis procedure, we first define the `toAsrt` function, which takes a symbolic state $\hat{\sigma} = (\hat{s}, \hat{H}, \hat{\pi})$, where $\hat{H} = (\hat{h}, \hat{P})$, and returns the corresponding assertion. The function is simple to implement: \hat{s} becomes a series of equalities, \hat{h} becomes a series of cell assertions, \hat{P} are lifted to predicate assertions and $\hat{\pi}$ is lifted to a pure assertion. Using `toAsrt`, we can also transform multiple symbolic states into an assertion by transforming them individually and gluing together the obtained assertions using disjunction.

We generate UX function specifications using the `synthesise` (Γ, f, P) algorithm, which takes a specification context Γ , a function $f(\vec{x}) \{ C; \text{return } E \}$ and its candidate pre-condition, $\vec{x} = \vec{x} \star P$, and uses bi-abduction to generate a set of UX specifications describing the behaviour of f starting from P . As $P = \text{emp}$ is a valid starting point, `synthesise` can be applied to any function without a priori knowledge. The `synthesise` algorithm is as follows:

1. Let $\hat{\theta} \triangleq \{\hat{x}/x \mid x \in \text{lv}(\vec{x} = \vec{x} \star P)\}$, $\hat{s} \triangleq \{x \mapsto \hat{x} \mid x \in \vec{x}\} \cup \{x \mapsto \text{null} \mid x \in \text{pv}(C) \setminus \vec{x}\}$, and $\hat{\sigma} = (\hat{s}, \emptyset, \text{true})$.
2. Add the symbolic representation of P to $\hat{\sigma}$: $\text{produce}(P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow \hat{\sigma}'$
3. Execute the function, obtaining a set of traces: $\hat{\sigma}', C, \text{ret} := E \Downarrow_{\Gamma}^{\text{bi}} \{(o_i, (\hat{\sigma}'_i, \hat{H}_i)) \mid i \in I\}$.
4. Then, for every obtained $(o_i, ((\hat{s}'_i, \hat{H}'_i, \hat{\pi}'_i), \hat{H}_i))$:
 - a. Complete the candidate pre-condition: $P_i \triangleq P \star \text{toAsrt}((\emptyset, \hat{H}_i, \text{true}))$.

⁵ This check is required due to our classical (linear) treatment of resource, appropriate for languages with explicit deallocation rather than garbage collection.

- b. Restrict the final store to the return/error variable: $\hat{s}_i'' \triangleq \hat{s}_i'|_{\{x\}}$, where $x = \text{ret}$ if $o_i = \text{ok}$ and $x = \text{err}$ otherwise.
- c. Create the post-condition: $Q_i \triangleq \text{toAsrt}(\hat{s}_i'', \hat{H}_i', \hat{\pi}_i')$.
- d. Return $[P_i] f(\vec{x}) [o_i : \exists \vec{y}. Q_i]$, where $\vec{y} \triangleq \text{lv}(Q_i) \setminus \text{lv}(P_i)$.

► **Theorem 8** (Correctness of synthesise).

$$[P'] f(\vec{x}) [o : \exists \vec{y}. Q] \in \text{synthesise}(\Gamma, f, P) \implies \Gamma \models [P'] f(\vec{x}) [o : \exists \vec{y}. Q]$$

► **Remark 9.** Step 4b corresponds to forgetting the local variables when moving from internal to external post-condition since symbolic states only have program variables in the store.

► **Remark 10.** Front-end heuristics to filter out “interesting” bugs, i.e., synthesised erroneous specification, can be easily implemented on top of our bi-abduction (e.g., filtering for manifest bugs as per Lee et al. [17]); for this paper, however, we are foremost interested in back-end engine development and therefore consider such front-end issues out of scope.

► **Remark 11.** Specifications with the same anti-frame can be coalesced into one via disjunction of their post-conditions. Moreover, if a specification does not branch on symbolic variables created by the execution of C , the pure part of the post-condition can be lifted to the precondition to create an EX specification [20], which can then be used both in OX verification and UX true bug-finding.

► **Remark 12.** Automatic predicate folding and unfolding may be required in some cases to prevent redundant fixes: e.g., if $y := g(); x := [y]$ and g has post-condition $\text{list}(\text{ret}, vs)$, the list predicate should be unfolded for the lookup to access the first value in the list. Gillian has heuristics-based automatic folding and unfolding, but we leave its description and the evaluation of its compatibility with bi-abduction for future work. Without automatic folding and unfolding, code must not break the interface barrier of the data structures it uses.

9 Evaluation

We have evaluated our CSE engine in the following two practical ways.

9.1 Companion Haskell Implementation

With our engine formalism, we have developed a companion Haskell implementation to demonstrate that the formalism is implementable (and to catch errors early by executing simple examples). The Haskell implementation follows the inference rules of $\hat{\sigma}, C \Downarrow_{\Gamma}^m o : \hat{\sigma}'$ given in §5, and the specific `consume` and `produce` operations in §6. We have implemented the search through the inference rules inside a symbolic execution monad, similar to other monads in the literature [7, 22]; the monad handles, e.g., demonic non-determinism (branching), angelic non-determinism (backtracking), per-branch state and global state.

9.2 Gillian OX and UX Compositional Analysis Platform

Our unified CSE engine took direct inspiration from the Gillian compositional OX platform. Using the ideas presented here, we returned to Gillian and adapted its CSE engine to handle both SL and ISL function specifications with real-world consume-produce implementations. Leveraging the identified difference between OX and UX reasoning, we were able to introduce UX reasoning to Gillian by adding, in essence, a OX/UX flag to the corresponding function. As these changes were isolated, existing analyses implemented in Gillian remain unaffected, including Gillian’s whole-program symbolic testing, previously evaluated on the Collections-C library [11], and Gillian’s compositional OX verification, previously evaluated on AWS

Table 1 Aggregated results of synthesising function specifications for the Collections-C library (commit 584e113). Results were obtained by setting the loop and recursive call unrolling limit to 3, on a MacBook Pro 2019 laptop with 16 GB memory and a 2.3 GHz Intel Core i9 CPU.

Library	Functions	GIL Inst.	Succ. Specs	Err. Specs	Time (s)
array	45	1784	251	260	1.36
deque	47	2312	271	210	2.25
hashset	14	160	7	112	9.43
hashtable	28	1527	31	147	15.67
list	66	2977	454	615	5.59
pqueue	10	557	90	51	3.96
queue	16	85	133	67	1.36
rbuf	9	181	9	17	0.07
slist	52	2269	292	1873	24.49
stack	16	85	136	88	0.50
treeset	17	214	28	106	0.36
treetable	36	1601	144	276	1.55
other	8	139	14	11	0.03
Total	364	13891	1860	3833	66.62

code [21]. In addition, we have implemented UX bi-abduction in Gillian, following the fixes-from-errors approach presented in §7, where functions are evaluated bottom-up along the call graph and previously generated specifications are used at call sites.

To evaluate Gillian’s new support for UX reasoning, we have tested its new UX bi-abduction analysis on real-world code, specifically, the Collections-C [25] data-structure library for C. As discussed in §8, specification synthesis using UX bi-abduction constitutes the back-end of Pulse-style bug-finding and is its most time-consuming part. The Collections-C library has 2.6K stars on GitHub and approximately 5.2K lines of code, and it uses many C constructs and idioms such as structures and pointer arithmetic. The data structures it provides include, e.g., dynamic arrays, linked lists, and hash tables. To carry out the evaluation, we extended previous work where the Gillian platform has been instantiated to the C programming language, called Gillian-C. Tbl. 1 presents the results of our new UX bi-abduction analysis, grouped by the data structures of the library: the numbers of associated functions; number of corresponding GIL instructions (GIL is the intermediate language used by Gillian); the number of success and error specifications; and the analysis time. Since one specification is synthesised per execution path, the number of specifications reflect the number of execution paths the Gillian engine was able to construct using bi-abduction. In summary, Gillian-C synthesises specifications for 364 functions of the Collections-C library, producing 5693 specifications in 66.92 seconds. We believe the results are promising both in terms of performance and number of specifications synthesised. One anomaly is that 58% of the execution time is spent on 3 of the 343 functions, leading to the creation of 1640 specifications. This anomaly arises because of the memory model currently in use by Gillian-C and not from a limitation of our formalisation or of the Gillian engine. More detailed analysis and a selection of generated specifications are available in [18].

10 Related Work

First-order Compositional Symbolic Execution. Static symbolic execution tools and frameworks based on first-order logic, such as CBMC [16] and Rosette [32, 27], can be made functionally compositional with respect to the variable store but not with respect to arbitrary state because they are not able to specify functions that manipulate memory in a way that would make the reasoning scalable.

Compositional Symbolic Execution. We work with a CSE engine with consume and produce operations, as found in, e.g., the OX tools VeriFast [14], Viper [23], and Gillian [11, 21]. In contrast, an alternative approach is to describe CSE inside a separation logic using proof search, as found in, e.g., Smallfoot [2, 3], Infer [4], and Infer-Pulse [17].

Here, we focus on formalising a CSE engine with consumers and producers, which more accurately models tool implementations. All the current consume-produce tools are based on OX reasoning. Some have detailed work on formalisation: Featherweight VeriFast [15] provides a Coq mechanisation inspired by VeriFast; Schwerhoff’s PhD thesis [30] and Zimmerman et al. [35] provide detailed accounts of Viper’s symbolic execution backend. Previous work has not, like us, introduced an axiomatic interface for their consume and produce operations. Because of the interface, our results are established using function specifications whose meaning is defined in standard SL/ISL-style, in particular, using the standard satisfaction relation for assertions defined independent of the choice of our CSE engine. This means that we can use specifications developed outside of our engine, e.g., using theorem provers, and vice versa. In contrast, the work on Featherweight VeriFast does not define an assertion satisfaction relation independent of their consume and produce operations. Schwerhoff does not give a soundness theorem at all. Lastly, Zimmerman et al. give a standard satisfaction relation (for implicit dynamic frames [31], a variant of SL [26]) but only embed this relation inside their concrete semantics instead of working with the standard definitions of function specifications (see Fig. 11 of their paper). Finally, we have demonstrated that our engine semantics provides a common foundation for OX and UX reasoning, with the difference in the underlying engine only amounting to the choice to use satisfiability or validity. This allows a straightforward extension of Gillian to support UX reasoning.

Bi-abduction. Bi-abduction was originally introduced for OX reasoning [5, 6] and led to Meta’s automatic Infer tool for bug-finding [4]. Recently, it was reworked for UX reasoning and led to Meta’s Infer-Pulse for true bug-finding [28, 17]. In these works, compositional symbolic execution is formalised using proof search, in the style of the Smallfoot description of symbolic execution [2, 3], with bi-abduction embedded into that proof search. In contrast, our UX bi-abduction is formulated as a separate layer on top of our CSE engine, establishing fixes from missing-resource errors using an idea introduced for OX bi-abduction by JaVerT 2.0 [10].

Alternating between OX and UX Reasoning. Smash by Godefroid et al. [13] is the most well-known tool to combine OX and UX reasoning. It is a first-order tool which “alternates” between OX and UX reasoning to speed up the program analysis implemented by the tool. However, citing Le et al. [17], who in turn report on personal communication with Godefroid, Smash-style analyses seem to have faced obstacles when put into practice in that they were “used in production at Microsoft, but are not used by default widely in their deployments, because other techniques were found which were better for fighting path explosion.” Our CSE engine, in contrast, has a completely different motivation in that its purpose is to host different types of OX and UX analyses.

Program Correctness and Incorrectness. We know of two program logics that work with both program correctness and incorrectness. Exact separation logic (ESL) [20] combines the guarantees of both SL and ISL, providing exact function specifications compatible with both OX verification and UX true bug-finding. Exact specifications are compatible with our CSE engine, e.g., in UX mode the engine can call exact unbounded function specifications of list algorithms and still preserve true bug-finding. Outcome logic (OL) [34] is based on OX

Hoare logic with a different approach to handling incorrectness based on the reachability of sets of states. No tool is currently based on OL; in the future, we hope to be able to extend straightforwardly our unified approach to incorporate OL.

11 Conclusions

We have introduced a compositional symbolic execution engine capable of creating and using function specifications arising from an underlying separation logic. Our engine is formally defined using a novel axiomatic interface which ensures a sound link between the execution engine and the function specifications using consume and produce operations. Thus, our engine creates function specifications usable by other tools, and uses function specifications from various sources, including theorem provers and pen-and-paper proofs. Additionally, we have captured the essence of the Gillian consume and produce implementations both operationally, using inference rules, and via an accompanying Haskell implementation, and shown that our operational description satisfies the properties of the axiomatic interface. In this way, we offer a degree of assurance that the real-world, heavily-optimised Gillian implementation is correct.

A surprising property of our semantics is that it provides a common foundation for both OX reasoning based on SL, and UX reasoning based on ISL. By leveraging the minimal differences between the OX and UX engines, we have extended the OX Gillian platform to support UX reasoning. This extension includes function specifications underpinned by ISL, enabling automatic true bug-finding using UX bi-abduction which our engine incorporates by creating fixes from missing-resource errors. We evaluate our extension using the Gillian instantiation to C, the first real-world tool to support both compositional correctness and incorrectness reasoning, grounded on a common formal compositional symbolic execution engine. Our instantiation preserves the previous OX verification evaluated on AWS code [21] and now automatically synthesises UX function specifications for the real-world Collections-C library using our UX bi-abduction technique.

We believe that our axiomatic interface and formalisation of UX bi-abduction serve as re-usable techniques, which we hope will provide valuable guidance for the implementation of the next-generation compositional symbolic execution engines.

References

- 1 Roberto Baldoni, Emilio Coppa, Daniele Cono D’elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys*, 51(3), 2018. doi:[10.1145/3182657](https://doi.org/10.1145/3182657).
- 2 Josh Berdine, Cristiano Calcagno, and Peter W. O’Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *International Conference on Formal Methods for Components and Objects*, 2005. doi:[10.1007/11804192_6](https://doi.org/10.1007/11804192_6).
- 3 Josh Berdine, Cristiano Calcagno, and Peter W. O’Hearn. Symbolic execution with separation logic. In *Asian Symposium on Programming Languages and Systems*, 2005. doi:[10.1007/11575467_5](https://doi.org/10.1007/11575467_5).
- 4 Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of C programs. In *NASA Formal Methods Symposium*, 2011. doi:[10.1007/978-3-642-20398-5_33](https://doi.org/10.1007/978-3-642-20398-5_33).
- 5 Cristiano Calcagno, Dino Distefano, Peter O’Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Principles of Programming Languages*, 2009. doi:[10.1145/1480881.1480917](https://doi.org/10.1145/1480881.1480917).

- 6 Cristiano Calcagno, Dino Distefano, Peter W. O’Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *Journal of the ACM*, 58(6), 2011. doi:10.1145/2049697.2049700.
- 7 David Darais, Nicholas Labich, Phúc C. Nguyen, and David Van Horn. Abstracting definitional interpreters (functional pearl). *Proceedings of the ACM on Programming Languages*, 1(ICFP), 2017. doi:10.1145/3110256.
- 8 José Fragoso Santos, Petar Maksimović, Théotime Grohens, Julian Dolby, and Philippa Gardner. Symbolic execution for JavaScript. In *Principles and Practice of Declarative Programming*, 2018. doi:10.1145/3236950.3236956.
- 9 José Fragoso Santos, Petar Maksimović, Daiva Naudžiūnienė, Thomas Wood, and Philippa Gardner. JaVerT: Javascript verification toolchain. *Proceedings of the ACM on Programming Languages*, 2(POPL), 2018. doi:10.1145/3158138.
- 10 José Fragoso Santos, Petar Maksimović, Gabriela Sampaio, and Philippa Gardner. JaVerT 2.0: Compositional symbolic execution for JavaScript. *Proceedings of the ACM on Programming Languages*, 3(POPL), 2019. doi:10.1145/3290379.
- 11 José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. Gillian, part I: A multi-language platform for symbolic execution. In *Programming Language Design and Implementation*, 2020. doi:10.1145/3385412.3386014.
- 12 Philippa Gardner, Sergio Maffeis, and Gareth David Smith. Towards a program logic for JavaScript. In *Principles of Programming Languages*, 2012. doi:10.1145/2103656.2103663.
- 13 Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. Compositional may-must program analysis: Unleashing the power of alternation. In *Principles of Programming Languages*, 2010. doi:10.1145/1706299.1706307.
- 14 Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods Symposium*, 2011. doi:10.1007/978-3-642-20398-5_4.
- 15 Bart Jacobs, Frédéric Vogels, and Frank Piessens. Featherweight VeriFast. *Logical Methods in Computer Science*, 11, 2015. doi:10.2168/LMCS-11(3:19)2015.
- 16 Daniel Kroening and Michael Tautschnig. CBMC – C bounded model checker. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2014. doi:10.1007/978-3-642-54862-8_26.
- 17 Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O’Hearn. Finding real bugs in big programs with incorrectness logic. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA1), 2022. doi:10.1145/3527325.
- 18 Andreas Lööw, Daniele Nantes-Sobrinho, Sacha-Élie Ayoun, Caroline Cronjäger, Petar Maksimović, and Philippa Gardner. Compositional symbolic execution for correctness and incorrectness reasoning (extended version), 2024. doi:10.48550/arXiv.2407.10838.
- 19 Andreas Lööw, Daniele Nantes-Sobrinho, Sacha-Élie Ayoun, Petar Maksimović, and Philippa Gardner. Matching plans for frame inference in compositional reasoning. In *European Conference on Object-Oriented Programming*, 2024. doi:10.4230/LIPIcs.ECOOP.2024.26.
- 20 Petar Maksimović, Caroline Cronjäger, Andreas Lööw, Julian Sutherland, and Philippa Gardner. Exact separation logic. In *European Conference on Object-Oriented Programming*, 2023. doi:10.4230/LIPIcs.ECOOP.2023.19.
- 21 Petar Maksimović, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. Gillian, part II: Real-world verification for JavaScript and C. In *Computer Aided Verification*, 2021. doi:10.1007/978-3-030-81688-9_38.
- 22 Adrian D. Mensing, Hendrik van Antwerpen, Casper Bach Poulsen, and Eelco Visser. From definitional interpreter to symbolic executor. In *International Workshop on Meta-Programming Techniques and Reflection*, 2019. doi:10.1145/3358502.3361269.
- 23 Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *Verification, Model Checking, and Abstract Interpretation*, 2016. doi:10.1007/978-3-662-49122-5_2.

- 24 Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs
that alter data structures. In *Computer Science Logic*, 2001. doi:10.1007/3-540-44802-0_1.
- 25 Srdja Panić. Collections-C: A library of generic data structures. <https://github.com/srdja/Collections-C>, 2014.
- 26 Matthew J. Parkinson and Alexander J. Summers. The relationship between separation
logic and implicit dynamic frames. In *European Symposium on Programming*, 2011. doi:
10.1007/978-3-642-19718-5_23.
- 27 Sorawee Porncharoenwase, Luke Nelson, Xi Wang, and Emina Torlak. A formal foundation
for symbolic evaluation with merging. *Proceedings of the ACM on Programming Languages*,
6(POPL), 2022. doi:10.1145/3498709.
- 28 Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter O'Hearn, and Jules Villard.
Local reasoning about the presence of bugs: Incorrectness separation logic. In *Computer Aided
Verification*, 2020. doi:10.1007/978-3-030-53291-8_14.
- 29 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in
Computer Science*, 2002. doi:10.1109/LICS.2002.1029817.
- 30 Malte Hermann Schwerhoff. *Advancing Automated, Permission-Based Program Verification
Using Symbolic Execution*. PhD thesis, ETH Zürich, 2016.
- 31 Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic
frames and separation logic. In *European Conference on Object-Oriented Programming
(ECOOP)*, 2009. doi:10.1007/978-3-642-03013-0_8.
- 32 Emina Torlak and Rastislav Bodík. A lightweight symbolic virtual machine for solver-aided
host languages. In *Conference on Programming Language Design and Implementation*, 2014.
doi:10.1145/2594291.2594340.
- 33 Hongseok Yang. *Local Reasoning for Stateful Programs*. PhD thesis, University of Illinois
Urbana-Champaign, 2001.
- 34 Noam Zilberstein, Derek Dreyer, and Alexandra Silva. Outcome logic: A unifying foundation for
correctness and incorrectness reasoning. *Proceedings of the ACM on Programming Languages*,
7(OOPSLA1), 2023. doi:10.1145/3586045.
- 35 Conrad Zimmerman, Jenna DiVincenzo, and Jonathan Aldrich. Sound gradual verification
with symbolic execution. *Proceedings of the ACM on Programming Languages*, 8(POPL), 2024.
doi:10.1145/3632927.

Matching Plans for Frame Inference in Compositional Reasoning

Andreas Lööw

Imperial College London, UK

Daniele Nantes-Sobrinho

Imperial College London, UK

Sacha-Élie Ayoun

Imperial College London, UK

Petar Maksimović

Imperial College London, UK

Runtime Verification Inc., Chicago, IL, USA

Philippa Gardner

Imperial College London, UK

Abstract

The use of function specifications to reason about function calls and the manipulation of user-defined predicates are two essential ingredients of modern compositional verification tools based on separation logic. To execute these operations successfully, these tools must be able to solve the frame inference problem, that is, to understand which parts of the state are relevant for the operation at hand. We introduce *matching plans*, a concept that is used in the Gillian verification platform to automate frame inference efficiently. We extract matching plans and their automation machinery from the Gillian implementation and present them in a tool-agnostic way, making the Gillian approach available to the broader verification community as a verification-tool design pattern.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Separation logic; Theory of computation → Automated reasoning

Keywords and phrases Compositional reasoning, separation logic, frame inference

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.26

Funding This work was supported by the EPSRC Fellowship “VetSpec: Verified Trustworthy Software Specification” (EP/R034567/1).

Acknowledgements We would like to thank José Fragoso Santos, who conceived the idea of matching plans (which we formalise in this work) as part of the work on JaVerT [6] and did their original implementation in Gillian. We would also like to thank the anonymous reviewers for their comments.

1 Introduction

Separation logic [18, 21] has enabled the verification community to develop analyses and tools that are *compositional* in the sense that they are able to analyse parts of the program in isolation and reuse the obtained results in broader contexts. Currently, some of the most prominent such tools are VeriFast [7], Viper [16], Gillian [13], and CN [20]. These tools achieve compositionality by being able to use function specifications at call sites instead of executing function bodies. In addition, to be able to reason about data structures such as lists and trees, the tools include support for user-defined inductive predicates that describe these data structures. When using function specifications and manipulating predicates, the tools have to be able to solve the *frame inference problem* [3], that is, understand which part of the state is relevant for the operation that is being performed. The ability to handle this problem efficiently is essential for their scalability and usability.

As part of building tools for compositional analysis, it is up to the tool designers to choose how to tackle frame inference, and the implications of the associated decisions should be understood closely as they affect both the tool implementation and the user experience. For example, two important tool-design questions are: “Which specification language should the tool use?” and “Is there a particular style in which the specifications should be written?”. Expectedly, the approaches in the literature are many and varied (which we discuss further in §7).

In this paper, we present the approach of the Gillian verification tool to the frame inference problem and show how the choices made allow for *efficient and predictable automation*. The approach captures and automates the *assertion-adaptation workflow* that users must follow to facilitate frame inference when working with tools that offer less automation, such as VeriFast and Viper. In more detail, for the assertions of function specifications and predicate definitions, Gillian automatically constructs a *matching plan* (MP), which provides:

1. (*efficiency*) an ordering of the subcomponents of each assertion (technically: the simple assertions) that guarantees that the associated frame inference will not backtrack, together with a description of how all associated free and existentially quantified variables can be learnt; and
2. (*predictability*) a clean separation between the structural and computational portions of frame inference.

MPs and their construction have not been described in depth before; we formalise both in a tool-agnostic way, thereby making the Gillian approach available to the broader verification community as a verification-tool design pattern.

The paper is structured as follows. We first introduce MPs informally using examples and discuss the key insights in §2. We then establish the required preliminaries in §3 and introduce MPs formally in §4, focusing on a core illustrative fragment of MPs as implemented in Gillian. Next, in §5, we show how to extend the MPs of §4 with more complex features that are available in Gillian. Finally, we conclude by evaluating the scalability and performance of the MP-based automation of Gillian (§6) and giving a detailed comparison with related work (§7).

2 Overview

We give an informal overview of *matching plans* (MPs), the key new concept we introduce in this paper. We first introduce the required background, which is the *consume/produce engine architecture* utilised by modern compositional symbolic execution tools, including VeriFast, Viper, and Gillian. Then, with the background in place, we introduce MPs using examples.

2.1 Background: Symbolic Execution Based on Consume and Produce

We place ourselves in the setting of semi-automated compositional verification tools based on symbolic execution [1] and separation logic (SL) and are underpinned by SMT solvers. In this context, the frame inference problem amounts to, given an assertion and a symbolic state, understanding which part of the symbolic state corresponds to the assertion. In particular, we are interested in tools such as VeriFast, Viper, and Gillian, implemented using *consumers and producers*, which are spatial variants of the, possibly more familiar, assert and assume, respectively. To *consume/produce* an assertion is to remove/add the corresponding spatial state from/to the current symbolic state and to assert/assume the pure constraints of the assertion. Our presentation focuses on consumption, as production does not require performing frame inference, and is therefore not of immediate interest. The two main use cases for frame inference in our setting are the following:

Use of function specifications to reason about function calls. Given a function specification $\{P\} f(\vec{x}) \{Q\}$, the verification tool *consumes* the part of the symbolic state that corresponds to the function pre-condition P (performing frame inference for P) and in its place *produces* a symbolic state that corresponds to the function post-condition Q .

Folding user-defined predicates. Given a predicate definition, folding a predicate consists of consuming the part of the symbolic state that corresponds to (a disjunct of) the definition and in its place producing the folded predicate, as discussed in more detail shortly.

Other use cases are similar. For example, reasoning about loops using loop invariants is largely similar to reasoning about function calls using function specifications.

2.2 Running Example: Folding a List Predicate

MPs help address two problems that arise during consumption and are related to frame inference: *the order of consumption* and *the learning of variables not given by context*, which we also refer to as *learning unknown variables*. We illustrate these problems using the example of folding the standard $\text{list}(x, vs)$ predicate that describes a singly-linked list starting at address x and carrying values vs . It is defined as follows, using standard SL notation, where “ \star ” denotes the separating conjunction and “ \mapsto ” denotes the cell assertion:

$$\begin{aligned} \text{list}(x, vs) \triangleq & (x = \text{null} \star vs = []) \vee \\ & (\exists v, x', vs'. x \mapsto v \star x + 1 \mapsto x' \star \text{list}(x', vs') \star vs = v : vs') \end{aligned}$$

This predicate has two disjuncts. The first disjunct states that the list is empty ($x = \text{null}$) and carries no values ($vs = []$). The second disjunct states that the list is non-empty, consisting of the list head node ($x \mapsto v \star x + 1 \mapsto x'$), which contains the node value, v , and the pointer to the next node, x' , and the tail of the list ($\text{list}(x', vs')$), while connecting the values appropriately ($vs = v : vs'$, meaning that vs is the result of prepending v to vs').

Let us now attempt to fold the predicate $\text{list}(x, vs)$ in the symbolic heap $\{1 \mapsto 1, 2 \mapsto 3, 3 \mapsto 2, 4 \mapsto \text{null}, 5 \mapsto 0, 6 \mapsto 1, 7 \mapsto 42\}$, knowing that $x = 5$.¹ As mentioned above, folding this list means performing frame inference by pinpointing a part of the symbolic state that corresponds to one of the predicate disjuncts. In consume/produce-based tools, this is done one *simple assertion* at a time, where an assertion is defined to be simple iff it does not contain the separating conjunction. Carving off the existential quantifiers, the first and second disjunct of the definition of $\text{list}(x, vs)$, respectively, have the following simple assertions:

$(\mathbf{A1}) \quad x = \text{null}$ $(\mathbf{A2}) \quad vs = []$	$(\mathbf{B1}) \quad x \mapsto v$ $(\mathbf{B2}) \quad x + 1 \mapsto x'$ $(\mathbf{B3}) \quad \text{list}(x', vs')$ $(\mathbf{B4}) \quad vs = v : vs'$
---	---

The first disjunct is relatively straightforward: to consume it means to check if it is possible for x to equal null and for vs to equal the empty list, which it is not since we know that $x = 5$. For the second disjunct, we additionally have to *learn* the values of the existentially quantified variables v , x' , and vs' . This can be more or less complex, depending on the *order* in which we process the assertions. For example, if we start with (B3), we will have to perform proof search, trying to guess the values of x' and vs' as they are not known, likely needing to backtrack and make different choices, which can be computationally expensive.

¹ For simplicity, in this example we describe symbolic heaps using cell assertions. In practice, one could choose to represent symbolic heaps differently for the purpose of, for example, efficient symbolic reasoning.

On the other hand, if we choose (B1) and (B2) first, given that we know $x = 5$, we could learn that $v = 0$ and $x' = 1$ trivially by inspecting the heap. From there, we can tackle (B3) by recursively folding the list $\text{list}(x', vs')$, ultimately learning $vs' = [1, 2]$, from which we can then process (B4), learning that $vs = [0, 1, 2]$. After having folded the predicate, the remaining frame is only the single heap cell $\{7 \mapsto 42\}$.

2.3 MPs for Predicate Folding and Function Calls

MPs provide a solution to the two problems illustrated in the previous section: given an assertion P , an MP for P provides an *ordering of the simple assertions of P* so that the consumption of P is guaranteed to not backtrack, as well as a *description of how free and existentially quantified variables of P can be learnt* during this consumption.

MPs are based on dividing parameters of assertions and predicates into *input parameters* (*ins*) and *output parameters* (*outs*). Intuitively, the *ins* of an assertion/predicate are the parameters that are sufficient to be provided so that the rest of the parameters, the *outs*, can be learned. For example, for the cell assertion $x \mapsto y$, if we know x we can learn y by looking it up in the heap: therefore, the *in* of the cell assertion is x and the *out* is y . For the $\text{list}(x, vs)$ predicate, on the other hand, the *in* is x and the *out* is vs .

Folding predicate example (running example). To give an example of an MP, consider again the second disjunct of the definition of the $\text{list}(x, vs)$ predicate:

$$x \mapsto v \star x + 1 \mapsto x' \star \text{list}(x', vs') \star vs = v : vs'$$

Assuming that only x is known before consumption, the MP for this disjunct is as follows (we give a formal definition of MPs in §4):

$$\begin{aligned} & [(x \mapsto v, [(v, O_1)]), \\ & (x + 1 \mapsto x', [(x', O_1)]), \\ & (\text{list}(x', vs'), [(vs', O_1)]), \\ & (vs = v : vs', [(vs, v : vs')])] \end{aligned}$$

which captures the following order of the simple assertions and ways of learning variables:

1. $x \mapsto v$ comes first, and from it we learn v as the cell assertion *out* by looking up the value corresponding to address x (which we know) in the heap, which is expressed using the placeholder variable O_1 ;
2. $x + 1 \mapsto x'$ comes next, and from it we learn x' again as the cell assertion *out*, noting that we know the assertion *in* $x + 1$ given that we know x ;
3. $\text{list}(x', vs')$ comes next, and from it we learn vs' as the predicate *out*, which can be done either by recursively folding as described above, or by matching against a predicate already existing in the symbolic state; and
4. $vs = v : vs'$ comes last, and from it we learn that vs equals $v : vs'$.

Function call example. We have exemplified MPs for folding predicate definitions. MPs are equally useful to handle function specifications. Creating an MP for a function specification amounts to creating an MP for the function pre-condition, which is effectively the same as creating an MP for a disjunct of a predicate. Interestingly, the use of *ins* and *outs* has as a consequence that MPs can be created only for function pre-conditions P in which all of the variables of P can be learnt if the function parameters are known. We have observed that this is not a restriction in practice, as the parameters are the only means that a function

can use to access or modify the state. In fact, a specification not obeying this property is likely either incorrect or contains resources not relevant for the function, which we can easily signal to the tool user.

2.4 MP-based Automation for Frame Inference

Gillian provides *predictable automation* for frame inference by providing machinery for automatically constructing MPs. This automation works by splitting the consumption process into two phases: a planning phase and a consumption phase – i.e., what we in the introduction of the paper referred to as “the structural and computational portions of frame inference” before having introduced consumption. An MP is automatically constructed in the planning phase. The consumption phase then follows the plan provided by the MP, which dictates consumption order and how variables are learnt. Because the planning phase is separate from the rest of consumption, planning is *predictable*. In particular, whereas the consumption phase relies on an unpredictable underlying SMT solver, the planning phase does not. The construction of MPs can therefore be understood (and, in particular, debugged) without having to take into consideration the more complicated consumption phase.

To compare Gillian with verification tools with no or little automation support for frame inference, e.g., the VeriFast tool: MPs can be said to capture the *assertion-adaptation workflow* tool users must follow when adapting assertions for such tools. Specifically, MPs capture this workflow by making clear the relationship between *ins* and *outs*. E.g., in VeriFast, the tool essentially requires that the MP can be directly “read off” assertions: the tool leaves it to the tool user to find both the consumption order and to “factor out” the *outs*, i.e., the parameters that will be learned during consumption given the *ins*. To exemplify, consider the simple assertion $x = 5$. Say x is unknown, an MP for this assertion can be directly read off the assertion: $[(x = 5, [(x, 5)])]$. Now, consider instead the simple assertion $y = x + z$ and say that y and z are known. An MP for this assertion is $[(y = x + z, [(x, y - z)])]$. In a tool without automation, the assertion would have to be adapted to $x = y - z$ such that how to learn the unknown variable x could be read off directly from the assertion. A slightly more complicated example is given by the simple assertion $x \mapsto x' + 1$ where x is known and x' is unknown. An MP for this assertion is $[(x \mapsto x' + 1, [(x', O_1 - 1)])]$, meaning that to adapt the assertion to a tool without automation, a user would have to introduce an intermediate variable as follows: $x \mapsto x'' \star x' = x'' - 1$. Similarly, in tools without automation, the consumption order must be specified by the user as well. E.g., in VeriFast assertions are consumed in left-to-right order. For example, consider the (contrived but simple) assertion $x > 5 \star x = 6$. Say that x is unknown, then there is no MP where $x > 5$ is consumed first, because x cannot be learnt from $x > 5$ without guessing. Instead, the only MP for the assertion is $[(x = 6, [(x, 6)]), (x > 5, [])]$. That is, without automation support, the user would have to switch the order of the simple assertions.

Gillian automates this workflow by automatically constructing MPs, thereby automating away assertion adaptations such as the adaptations exemplified above. Moreover, the automation helps in using assertions generated by other tools (which do not necessarily generate assertions in the style verification tools expect). In §4, we provide a formal description of a simple planning algorithm that is able to construct MPs for assertions with unknown variables embedded inside simple arithmetic expressions. This simple planning algorithm is the theoretical core of the MPs and the MP-based automation of Gillian. This simple core provides a foundation that can be extended in multiple directions. We discuss some of those extensions in §5, including an example of extending the learning algorithm to handle an instance of list-based learning, which originates from a large verification case study that has been carried out in Gillian where the automation eased the amount of assertion adaptation needed.

3 Preliminaries: Assertion Language

As mentioned in the introduction, we introduce the formal description of MPs in two steps. First, in §4, we formalise a simple version of MPs, which we call core MPs. Second, in §5, we discuss extensions of core MPs that widen their applicability. In this section, we formally define the simple assertion language we use to formalise core MPs. In particular, the simple assertion language we introduce here is for the simple memory model commonly used in theoretical investigations into separation logic. When we discuss extensions of core MPs in §5, we show that core MPs are easily extended to other memory models.

Given a set of logical variables $x, y, z, \dots \in \text{LVar}$, the syntax of our assertion language is as follows:

► **Definition 1** (Syntax of Assertions).

$$\begin{aligned} v \in \text{Val} &\triangleq n \in \text{Int} \mid b \in \text{Bool} \mid \text{null} \mid [\vec{v}] \\ E \in \text{Exp} &\triangleq v \mid x \mid \neg E \mid E_1 \wedge E_2 \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 : E_2 \mid E_1 < E_2 \mid E_1 = E_2 \\ P \in \text{Asrt} &\triangleq E \mid \text{emp} \mid E_1 \mapsto E_2 \mid P_1 \star P_2 \mid p(\vec{E}_1; \vec{E}_2) \end{aligned}$$

The values, Val , consist of integers, Booleans, null values, and lists of values. (Note that we will use the notation $[x]$ to denote *both* lists with elements x and the type of lists with elements of type x . E.g., $[\text{LVar}]$ denotes the type of lists of LVar s.) The expressions, Exp , are standard, including a representative selection of operators. We do not include program variables, as they are not needed for our discussion here; they can be treated straightforwardly. The assertions, Asrt , are also standard, except that predicate assertions, $p(\vec{E}_1; \vec{E}_2)$, have their arguments separated into *ins* and *outs*, which are used to construct MPs for predicates.

Definitions of predicates (e.g., list from the overview section) come from a set Preds :

► **Definition 2** (Syntax of Predicates). *We describe the predicate definitions of Preds using the following syntax:*

$$p(\vec{x}_{in}; \vec{x}_{out}) = \bigvee_{i=1}^n (\exists \vec{x}_i. P_i)$$

where $p \in \text{Str}$ (strings), $\vec{x}_{in}, \vec{x}_{out}, \vec{x}_i \in [\text{LVar}]$, and $P_i \in \text{Asrt}$.² Predicates abide by the following restrictions: \vec{x}_{in} and \vec{x}_{out} have no duplicates; \vec{x}_{in} and \vec{x}_{out} are disjoint and for every $i \in [1, n]$, $\vec{x}_{in} \cup \vec{x}_{out}$ and \vec{x}_i are disjoint; and P_i only has logical variables from $\vec{x}_{in} \cup \vec{x}_{out} \cup \vec{x}_i$.

The semantics of assertions is standard. Let $h : \text{Nat} \rightarrow_{fin} \text{Val}$ (with $\text{Nat} \subset \text{Int}$) denote a heap and $\theta : \text{LVar} \rightarrow_{fin} \text{Val}$ a logical interpretation, and let $\llbracket E \rrbracket_\theta$ be the standard expression evaluation function. With these in place, the semantics of assertions is as follows:

► **Definition 3** (Semantics of Assertions). *The satisfaction relation for assertions, denoted by $\theta, h \models P$, is defined as follows:*

$$\begin{aligned} \theta, h \models E &\Leftrightarrow \llbracket E \rrbracket_\theta = \text{true} \wedge h = \emptyset \\ \text{emp} &\Leftrightarrow h = \emptyset \\ E_1 \mapsto E_2 &\Leftrightarrow h = \{\llbracket E_1 \rrbracket_\theta \mapsto \llbracket E_2 \rrbracket_\theta\} \\ P_1 \star P_2 &\Leftrightarrow \exists h_1, h_2. h = h_1 \uplus h_2 \wedge \theta, h_1 \models P_1 \wedge \theta, h_2 \models P_2 \\ p(\vec{E}_1; \vec{E}_2) &\Leftrightarrow \exists i. \exists \vec{v}_i. \theta[\vec{x}_{in} \mapsto \llbracket \vec{E}_1 \rrbracket_\theta, \vec{x}_{out} \mapsto \llbracket \vec{E}_2 \rrbracket_\theta, \vec{x}_i \mapsto \vec{v}_i], h \models P_i \\ &\quad \text{for } p(\vec{x}_{in}; \vec{x}_{out}) = \bigvee_{i=1}^n (\exists \vec{x}_i. P_i) \in \text{Preds} \end{aligned}$$

² Formally, Preds is a set with elements of type $(\text{Str}, [\text{LVar}], [\text{LVar}], ([\text{LVar}], \text{Asrt}))$, but this is not important for our development.

We need the following definitions in our discussion on MPs. As discussed in the overview, MPs are defined over collections of *simple assertions*:

► **Definition 4** (Simple Assertion). *An assertion P is simple iff it syntactically contains no separating conjunction: e.g., $x \mapsto 5$ is simple, and so is $\text{foo}(x; y, z)$ regardless of how foo is defined, but $x \mapsto 0 \star y \mapsto 0$ is not simple.*

We will also need to talk about the *free logical variables of expressions and assertions*:

► **Definition 5** (Free Logical Variables of Expressions and Assertions). *We write $\text{lv}(E)$ to denote the free logical variables of an expression E and extend this notation to lists of expressions, writing $\text{lv}(\vec{E})$. The function lv naturally extends to assertions.*

4 Formalisation of Core MPs

We now formally describe a simple version of MPs, which we call core MPs, for the assertion language introduced in the previous section. We discuss extensions of core MPs in the next section.

4.1 Formal Definition of MPs

MPs are defined w.r.t. a given *knowledge base* KB and an assertion P . A knowledge base is a set of the currently known logical variables, which grows during planning. MPs have type $[(\text{Asrt}, [(\text{LVar}, \text{Exp})])]$ and are defined as follows:

► **Definition 6** (Matching Plans (MPs)). *Given a knowledge base KB and an assertion P of the form $P_1 \star \dots \star P_n$ where $P_i|_{i=1}^n$ are simple assertions, MP is a matching plan for P with respect to KB iff $\text{plan}(KB, [P_1, \dots, P_n], MP)$, as per the rules in Fig. 1 and Fig. 2.*

As a shorthand, we sometimes say that an assertion that has an MP is “plannable” (where the KB is usually left to be implied by context).

The rules in Fig. 1 and Fig. 2 are designed to ensure that if the simple assertions of P are consumed in the order specified by an MP of P , then the *ins* of each simple assertion P will be known before the simple assertion is consumed. To say this more formally, let us denote by $\text{ins}(KB, P)$ the *ins* of the assertion P under the knowledge base KB . Now, if $\text{plan}(KB, [P_1, \dots, P_n], [(P_{m_i}, _)|_{i=1}^n])$ holds, then $[P_{m_i}|_{i=1}^n]$ is a permutation of $[P_i|_{i=1}^n]$, and if we let KB_i denote the knowledge base before the i -th iteration of the planning, then for every $1 \leq i \leq n$, it holds that $\text{ins}(KB_i, P_{m_i}) \subseteq KB_i$.

We now explain the rules in Fig. 1 and Fig. 2. We go by bottom-up order, starting with the rules for expressions.

Explanation of expression planning rules. Fig. 1 contains the rules for expression planning. The entry point into expression planning is the `planExps` relation. Here we discuss planning of a single expression, that is, the relation `planExp`, and return to the non-single case when discussing assertion planning. For core MPs, we only include a few basic learning rules for `planExp` to illustrate the basic idea. The rule (`PURE`) state that expressions where all variables are known are trivially plannable. The rule (`PURE-EQ`) is more interesting. It says, given an equality expression where all variables of one side are known, the expression is plannable if the unknown variables of the other side of the expression can be factored out. The factoring is done by the `learnEq(KB, Ek, Eu)` function, where E_k is known and E_u is unknown. This function, at a high level, tries to move known parts of E_u to E_k until only a

$$\begin{array}{c}
 (\text{LIST-BASE}) \frac{}{\text{planExps}(KB, [], [])} \\
 \\
 (\text{LIST-IND}) \frac{1 \leq i \leq n \quad \text{planExp}(KB, E_i, [(x_j, E_{i_j})|_{j=1}^k]) \quad \text{planExps}(KB', [E_1, \dots, E_{i-1}, E_{i+1}, \dots, E_n], res')}{\text{planExps}(KB, [E_1, \dots, E_n], [(x_j, E_{i_j})|_{j=1}^k] ++ res')}
 \end{array}$$

$$\begin{array}{c}
 (\text{PURE}) \frac{1v(E) \subseteq KB}{\text{planExp}(KB, E, [])} \quad (\text{PURE-EQ}) \frac{1v(E_i) \subseteq KB \quad 1v(E_j) \not\subseteq KB \quad \{i, j\} = \{1, 2\} \quad \text{learnEq}(KB, E_i, E_j) = res}{\text{planExp}(KB, E_1 = E_2, res)}
 \end{array}$$

$$\begin{aligned}
 \text{learnEq}(KB, E_k, x) &\triangleq [(x, E_k)] \\
 \text{learnEq}(KB, E_k, \neg E) &\triangleq \text{learnEq}(KB, \neg E_k, E) \\
 \text{learnEq}(KB, E_k, E_1 + E_2) &\triangleq \begin{cases} \text{learnEq}(KB, E_k - E_1, E_2), & \text{if } 1v(E_1) \subseteq KB, 1v(E_2) \not\subseteq KB \\ \text{learnEq}(KB, E_k - E_2, E_1), & \text{if } 1v(E_1) \not\subseteq KB, 1v(E_2) \subseteq KB \end{cases} \\
 \text{learnEq}(KB, E_k, E_1 - E_2) &\triangleq \begin{cases} \text{learnEq}(KB, E_1 - E_k, E_2), & \text{if } 1v(E_1) \subseteq KB, 1v(E_2) \not\subseteq KB \\ \text{learnEq}(KB, E_k + E_2, E_1), & \text{if } 1v(E_1) \not\subseteq KB, 1v(E_2) \subseteq KB \end{cases}
 \end{aligned}$$

 **Figure 1** Rules: `planExps`, `planExp`, and `learnEq` for expressions.

logical variable is left, which can then be learnt. The function `learnEq` returns a list since with support for lists in the expression language it is possible to learn multiple variables from one expression. We do not include learning rules for lists here but discuss a list-related extension in §5. Note that including learning rules for different operators is always optional: learning rules are only required to use operators in learning (i.e., to enable more automation), operators without special learning rules can still be planned as long as all unknown variables of the input assertion can be learnt elsewhere.

► **Example 7.** Say, $KB = \{x, y\}$. Some simple examples include

- $\text{learnEq}(KB, x, z) = [(z, x)]$,
- $\text{learnEq}(KB, x, z + 5) = \text{learnEq}(KB, x - 5, z) = [(z, x - 5)]$,
- $\text{planExp}(KB, x = z, [(z, x)])$,
- $\text{planExp}(KB, x = \neg z, [(z, \neg x)])$, and
- $\text{planExp}(KB, x = z + 5 - y, [(z, x + y - 5)])$.

Explanation of assertion planning rules. Fig. 2 contains the rules for assertion planning. We first discuss `planSimple`, which is the planning relation for simple assertions. Pure simple assertions are handled by the (`CONJ`) rule. The rule takes a list of expressions formed by the conjuncts of the input expression. It does so by relying on the `planExps` relation for planning of lists of expressions: the relation specifies expression orders for which all `outs` can be learned (its definition is similar to the definition of `plan`, which we discuss shortly). Non-conjunct pure simple assertions are covered by the degenerate case of the (`CONJ`) rule when $n = 1$. The (`EMP`) rule is trivial since `emp` is always plannable. The (`HEAP`) rule for cell assertions $E_1 \mapsto E_2$ states that a cell assertion is plannable if we (at least) know its `ins`, that is, the

$$\begin{array}{c}
 (\text{PLAN-BASE}) \frac{}{\text{plan}(KB, [], [])} \\
 \\
 (\text{PLAN-IND}) \frac{1 \leq i \leq n \quad \text{planSimple}(KB, P_i, [(x_j, E_j)|_{j=1}^k])}{\begin{array}{l} KB' \triangleq KB \cup \{x_j|_{j=1}^k\} \\ \text{plan}(KB', [P_1, \dots, P_{i-1}, P_{i+1}, \dots, P_n], MP) \end{array}} \frac{}{\text{plan}(KB, [P_1, \dots, P_n], (P_i, [(x_j, E_j)|_{j=1}^k]): MP)} \\
 \\
 \hline
 \\
 (\text{CONJ}) \frac{n \geq 1 \quad \text{planExps}(KB, [E_1, \dots, E_n], res)}{\text{planSimple}(KB, \wedge_{i=1}^n E_i, res)} \\
 \\
 (\text{EMP}) \frac{}{\text{planSimple}(KB, \text{emp}, [])} \quad (\text{HEAP}) \frac{\text{planExps}(KB \cup \{O_1\}, [E_2 = O_1], res)}{\text{planSimple}(KB, E_1 \mapsto E_2, res)} \\
 \\
 (\text{PRED}) \frac{\begin{array}{l} \text{lv}(E_1) \subseteq KB \\ \vec{E}_2 = [E_{2i}|_{i=1}^n] \\ \text{planExps}(KB \cup \{O_1, \dots, O_n\}, [E_{21} = O_1, \dots, E_{2n} = O_n], res) \end{array}}{\text{planSimple}(KB, p(\vec{E}_1; \vec{E}_2), res)}
 \end{array}$$

■ **Figure 2** Rules: `plan` and `planSimple` for assertions.

logical variables of E_1 , and E_2 is plannable according to `planExps`. We also need to record that the *out* of the cell assertion, that is, E_2 , equals the contents of the cell at E_1 in memory, and from this equality we may be able to learn further information. Note, however, that the heap is not available during the planning process, and we therefore use a placeholder variable O_1 , which is a reserved logical variable that is not allowed to occur in assertions, that will be instantiated to the actual heap contents at runtime (see Ex. 8). Finally, the (PRED) rule generalises the planning of cell assertions to predicates by abstracting all of the predicate *outs* using placeholder variables O_1, \dots, O_n to then be instantiated and linked to $E_{2i}|_{i=1}^n$ appropriately.

► **Example 8.** To illustrate how placeholder variables work in practice, consider consuming $x \mapsto y$ knowing that $x = 41$. An MP for this assertion, by (HEAP), is $[(x \mapsto y, [(y, O_1)])]$. Say the current heap has a cell $41 \mapsto 42$. In this case, at the time of consumption the placeholder variable O_1 will be instantiated with the contents of the cell at x , which equals 42, yielding $y = 42$.

We now turn to the main entry point: the `plan` relation. The main rule of `plan`, the (PLAN-IND) rule, specifies valid orders of the simple assertions of P that guarantee the learning of all their *outs*, extending the knowledge base as the *outs* of each simple assertion are learnt. At each choice point, the rule is applicable for a simple assertion P_i whose *ins* are all known using the `planSimple` relation, together with all the logical variables $x_j|_{j=1}^k$ that can be learnt from P_i and expressions $E_j|_{j=1}^k$ describing how they can be learnt. The rule then extends the knowledge base with learnt variables, inductively repeats the planning for the remaining simple assertions, and finally adds the result of `planSimple`, $(P_i, [(x_j, E_j)|_{j=1}^k])$, to the full MP.

► **Example 9.** Let $P = \text{list}(w; vs) \star x \mapsto y \star z \mapsto w \star z = y + 21$ and $KB = \{x\}$, and let us use the provided rules to construct an MP for P . Branching on the (PLAN-IND) rule, we have that for $i = 1$ and $i = 3$ we cannot apply (PRED) and (HEAP) as we do not know all the

corresponding ins (in particular, w and z), and for $i = 4$ we cannot apply (PURE-EQ) as we do not know all of the variables on either side of the equality. For $i = 2$, however, we can apply (HEAP) as we know x and we have $\text{planExps}(KB \cup \{O_1\}, [y = O_1], [(y, O_1)])$, which follows from $\text{planExp}(KB \cup \{O_1\}, y = O_1, [(y, O_1)])$, which in turn follows from $\text{learnEq}(KB \cup \{O_1\}, O_1, y) = [(y, O_1)]$. Returning to (PLAN-IND) and continuing until the end, we obtain the following MP for P :

$$[(x \mapsto y, [(y, O_1)]), (z = y + 21, [(z, y + 21)]), (z \mapsto w, [(w, O_1)]), (\text{list}(w; vs), [(vs, O_1)])]$$

4.2 Computing MPs

Given the inference-rule formalisation of MPs in the previous section, it is easy to construct an algorithm for automatically constructing MPs: a simple greedy algorithm searching through the `plan` relation is sufficient to find an MP for a given assertion. We can greedily pick the first valid choice we find at each choice point of the rules of the `plan` relation and its auxiliary relations. That is, no backtracking is needed to explore multiple choice points (note that here we are referring to backtracking during the construction of MPs, not the backtracking during consumption that MPs help to avoid as discussed earlier). This is because *outs* are only learnt by equality reasoning and therefore learning only happens when forced, so the order in which *outs* are learnt does not matter. In §6, we report performance numbers of this simple greedy algorithm as implemented in Gillian.

Note that no soundness result is needed for MPs to ensure soundness for the verification tool as a whole: as long as no simple assertions are dropped from a given input assertion, it is not possible to construct an “incorrect” MP that leads to an unsoundness bug in the verification tool. This is because an incorrectly constructed MP will simply make the consumption following the MP to fail and force the verification process to abort. To exemplify, consider the assertion $x = 1$ with an empty knowledge base. Say we construct the incorrect MP $[(x = 1, [(x, 0)])]$, suggesting to instantiate x to 0. During consumption, this MP will lead to $0 = 1$ being consumed, which will of course fail. Similarly, an MP missing one or more *outs* will cause the consumption to fail as well. E.g., an incorrect MP $[(x = 1, [])]$ for the same assertion, where the x variable is missing, will be caught during consumption as well since x will be left uninstantiated.

4.3 MPs for Function Specifications and Predicates

Given the definition of an MP for an assertion, we can easily define MPs for function specifications and predicates, as we now explain and exemplify.

MPs for function specifications are defined as follows:

- ▶ **Definition 10** (Matching Plans: Function Specification). *An MP for a function specification $\{\vec{x} = \vec{x} \star P\} f(\vec{x}) \{Q\}$ is an MP for P with knowledge base $\{\vec{x}\}$.*

For function specifications of the above form, where the function parameters \vec{x} are bound to logical variables \vec{x} , when symbolically executing a function call, the values of \vec{x} are given by the arguments provided in the call, and \vec{x} can therefore be assumed to be known at the start of the planning.³

MPs for predicates are defined as follows:

³ As we do not include program variables in assertions, pre-conditions are formally pairs of the form (\vec{x}, P) , but we stylise them to remain in line with the usual SL syntax.

► **Definition 11** (Matching Plans: Predicates). An MP for a predicate $p(\vec{x}_{in}; \vec{x}_{out}) = \bigvee_{i=1}^n (\exists \vec{x}_i. P_i)$, is a list of MPs, $[mp_i]_{i=1}^n$, such that mp_i is an MP for P_i with knowledge base $\{\vec{x}_{in}\}$.

Recall that the use case for MPs for predicates is predicate folding: to fold a predicate $p(\vec{x}_{in}; \vec{x}_{out})$ we have to know its *ins* \vec{x}_{in} , whereas the existentials from the predicate body disjuncts need to be inferable from these *ins* and the *outs* \vec{x}_{out} can be either provided or optionally left to be inferred from the *ins*. Also note that how MPs are defined for predicates does not depend on how much folding automation the verification tool provides: from the perspective of planning, it does not matter if the fold was requested manually or automatically. Lastly note that because the *ins* and *outs* of a predicate are given at the time of definition, failure to construct an MP can be reported early, i.e., at the time of definition, rather than when the predicate is used in folding.

Examples of plannable predicates include all predicates for standard data structures. We discuss some data-structure examples in more detail below.

► **Example 12.** We return to the standard SL predicate $list(x; vs)$ from our running example, where we now have separated the *ins* and *outs*. Recall, the predicate is defined as follows:

$$\begin{aligned} list(x; vs) \triangleq & (x = \text{null} \star vs = []) \vee \\ & (\exists v, x', vs'. list(x'; vs') \star x \mapsto v \star x + 1 \mapsto x' \star vs = v : vs') \end{aligned}$$

Importantly, despite the fact that the definition of the *list* predicate is recursive, no recursion is needed to express (or compute) the MP for the predicate. Per Def. 11, the predicate is plannable since the following is an MP for the predicate:

$$\begin{aligned} [[(x = \text{null}, & [(x, \text{null})])], \\ & (vs = [], & [(vs, [])])], \\ & [(x \mapsto v, & [(v, O_1)])], \\ & (x + 1 \mapsto x', & [(x', O_1)])], \\ & (list(x'; vs'), & [(vs', O_1)])], \\ & (vs = v : vs', & [(vs, v : vs')])]] \end{aligned}$$

where the first element of the list is an MP for the first disjunct of the predicate body (the null disjunct) and the second element of the list is an MP for the second disjunct of the predicate body (the non-null disjunct).

► **Example 13.** We easily see that the following two variants of the singly-linked list predicate *list* and the doubly-linked list predicate *dlist* are plannable:

$$\begin{aligned} list(x) \triangleq & (x = \text{null}) \vee (\exists v, x'. x \mapsto v, x' \star list(x')) \\ list(x; n) \triangleq & (x = \text{null} \star n = 0) \vee (\exists v, x'. x \mapsto v, x' \star list(x'; n - 1)) \\ \text{dlseg}(x, x', y, y'; vs) \triangleq & (vs = [] \star x = x' \star y = y') \vee \\ & (\exists x'', v, vs'. vs = v : vs' \star x \mapsto v, x'', y' \star \text{dlseg}(x'', x', y, x; vs')) \\ \text{dlist}(x, y; vs) \triangleq & \text{dlseg}(x, \text{null}, y, \text{null}; vs) \end{aligned}$$

► **Example 14.** For a non-list example of a plannable data-structure predicate, we turn to binary search trees (extending our simple assertion language's values and expressions with support for sets):

$$\begin{aligned} bst(x; vs) \triangleq & (x = \text{null} \star vs = \emptyset) \vee \\ & (\exists v, l, r, vs_l, vs_r. x \mapsto v, l, r \star bst(r; vs_r) \star bst(l; vs_l) \star \\ & vs = vs_l \uplus \{v\} \uplus vs_r \star vs_l < v \star v < vs_r) \end{aligned}$$

► **Example 15.** Finally, we highlight that it is up to the tool user to make sensible choices for *ins* and *outs*, as not all plannable choices need be equally useful in practice. Note that this is not a requirement introduced by MPs, rather, MPs simply make this requirement explicit by separating *ins* from *outs*. To illustrate, consider the standard acyclic- and cyclic-list-segment predicates:

$$\begin{aligned}\text{lseg}(x, y, vs) &\triangleq (x = y \star vs = []) \vee \\ &\quad (\exists x', v, vs'. x \neq y \star x \mapsto v, x' \star vs = v : vs' \star \text{lseg}(x', y, vs')) \\ \text{cseg}(x, y, vs) &\triangleq (x = y \star vs = []) \vee \\ &\quad (\exists x', v, vs'. x \mapsto v, x' \star vs = v : vs' \star \text{cseg}(x', y, vs'))\end{aligned}$$

where the only difference between the two is in that the former does not allow the start and the end pointers to be equal in the second disjunct of its definition (specified by $x \neq y$) and the latter does not have this constraint, and consider the various choices of *ins* and *outs*, with the goal being that the *ins* should uniquely determine the *outs*, minimising potential branching coming from folding. For both *lseg* and *cseg*, x has to be an *in*, as otherwise, given that the list is singly-linked (forward-pointing), we would have no way of determining where the list segment starts. Observe that only having x as an *in* is enough for both disjuncts in both predicate definitions to be plannable. However, without additional *ins*, we do not know where the list segment ends, and folding the predicate would yield up to n branches, where n is the length of the maximal list segment in the heap starting from x . Adding y as an *in* solves this issue for *lseg*, since then we fix the list segment by knowing both the start and its end; similarly, we could add vs as an *in* and then we would know the length of the list segment, which, together with its start, would also uniquely determine it. Interestingly, adding y as an *in* for *cseg* still does not uniquely determine the cyclic list segment, as its two disjuncts are not disjoint: for example, in the heap $\{42 \mapsto 0, 42\}$, we could fold both *cseg*(42, 42; []) and *cseg*(42, 42; [0]). Adding vs as an *in* of *cseg*, however, does solve the issue, as the length of the list segment then becomes unambiguous. The same situation would come up with any predicate whose disjuncts are not disjoint.

5 Extensions

Having formalised core MPs in the previous section, we now discuss important MP extensions that widen the applicability and usefulness of MPs. The extensions we discuss here are from the implementation of MPs in the Gillian tool.

Parametric matching plans. To support multiple programming languages (e.g., C and JavaScript), Gillian is parametric on the memory model used for analysis. In supporting parametricity, Gillian's implementation of MPs is parametric as well, which we now show is a simple extension of core MPs.

Memory models in Gillian are described in terms of *core predicates*, which represent the fundamental units of the memory model. Core predicates are described using core-predicate assertions with syntax $c(\vec{E}_1; \vec{E}_2)$, where $c \in \text{Str}$ is the name of the core predicate and \vec{E}_1 and \vec{E}_2 are the *ins* and *outs* of the core predicate. Each memory model instance must provide a set of core predicates and the *ins* and *outs* of each core predicate. For example, for the simple memory model we used for core MPs, the only core predicate is the cell assertion, $E_1 \mapsto E_2$, which has E_1 as an *in* and E_2 as an *out* – or, more formally: $\mapsto(E_1; E_2)$. Another example is given by the Gillian C memory model, whose core predicates include a cell core predicate of the form $(E_b, E_o) \mapsto E_v$, which states that the cell at offset E_o in the block at location E_b has contents E_v , where E_b and E_o are the *ins* and E_v is an *out*, and a block-bound predicate $\text{bound}(E_b; n)$, which states that the block at location E_b has length n .

From the discussion above, it is straightforward to generalise planning to parametric planning since core-predicate assertions $c(\vec{E}_1; \vec{E}_2)$ share syntax with user-defined-predicate assertions $p(\vec{E}_1; \vec{E}_2)$ and therefore for the purpose of planning are the same. That is, the (PRED) rule of Fig. 2 can be used to plan core-predicate assertions. Indeed, recall that for the simple memory model we used for core MPs, the (HEAP) rule is indeed a special case of the (PRED) rule (see Fig. 2).

Extending learning capabilities. In some large verification projects, it might be desirable to extend the learning capabilities of the core MP algorithm with project-custom learning rules: for example, to avoid repetitive manual project-specific massage of assertions to make them plannable with respect to the simple learning rules of core MPs.

We discuss one such learning extension that has been implemented in Gillian, specifically, a list-related extension that was added for the largest case study carried out in Gillian: the verification of C and JavaScript implementations of the deserialisation module of the AWS Encryption SDK message header [13]. To illustrate, consider the assertion $P \triangleq a = a_l ++ a_r \star \text{len}(a_l) = l$ with $KB = \{a, l\}$, where $++$ denotes list concatenation and len denotes list length. P is not plannable using the core MP algorithm, because the algorithm can only learn logical variables: as list length is not injective, a_l cannot be learned from l and planning is stuck. However, P becomes plannable if knowledge bases are allowed to also contain expressions of the form $\text{len}(x)$: $\text{len}(a_l)$ can then be learnt from $\text{len}(a_l) = l$, and both a_l and a_r can be learnt, respectively, as $a_l = a[0 : \text{len}(a_l)]$ and $a_r = a[\text{len}(a_l) : \text{len}(a)]$ from $a = a_l ++ a_r$, where $E_1[E_2 : E_3]$ denotes list slicing from index E_2 inclusive to index E_3 exclusive.

The above example may look simple but was essential for creating MPs of predicates describing the data structures used in the AWS case study. At a high level, AWS Encryption SDK message headers are buffers (arrays of bytes) that comprise a number of sections, with each section having either a static length described by the standard or a dynamic length derived from content appearing in the earlier sections of the buffer. In that context, the list-length extension allowed for clear definitions that follow the descriptions in their official documentation. Otherwise, the predicates would have to be stated using more complex operators. Specifically, using Gillian notation, (part of) the predicate describing the message header is as follows:⁴

```
pred Header(+rawHeader, ver, type, sId, msgId, ECLen, ECKs, ...) :
  rawHeader == ([ver, type] ++ #rawSId ++ msgId ++ #rawECLen ++ #EC ++ ...) *
  len(#rawSId) == 2 * UInt16(#rawSId, suiteId) * len(msgId) == 16 *
  len(#rawECLen) == 2 * UInt16(#rawECLen, ECLEN) *
  len(#EC) == ECLEN * EncryptionContext(#EC, ECKs) * ...
```

while without the list-length extension its definition would be as follows:

```
pred Header(+rawHeader, ver, type, sId, msgId, ECLen, ECKs, ...) :
  [ver, type] == rawHeader[0, 2] * #rawSId == rawHeader[2, 4] *
  UInt16(#rawSId, suiteId) * msgId == rawHeader[4, 16] *
  #rawECLen = rawHeader[20, 22] * UInt16(#rawECLen, ECLEN) *
  #EC == rawHeader[22, 22 + ECLEN] * EncryptionContext(#EC, ECKs) * ...
```

⁴ In Gillian notation, the `+` symbol denotes a predicate `in`, and the `#` symbol denotes existential quantification. The `UInt16(+x, y)` predicate states that the two bytes given by `x` can be viewed as an unsigned 16-bit integer `y`, while the `EncryptionContext` predicates is specific to the AWS case study and its meaning is not relevant here.

By comparing these two definitions, we can see that not only is the latter more difficult to read and understand, but is also more error-prone, as the list-slicing indices get progressively more complicated.

This extension approach is not limited to the above list-length example and can be applied for other expressions: e.g., we might choose to keep $a + b$ in the KB if we know it but do not know either a or b . These further extensions can be added on an as-needed basis straightforwardly by modifying the OCaml code of Gillian. An interesting direction for the future is to develop a small domain-specific language for MP rules (i.e., rules like those in e.g. Fig. 1) to simplify extending Gillian’s MP algorithm with new rules for extended learning capabilities.

Support for magic wands. MPs can easily be extended to support the magic wand operator $\rightarrow*$. Formally, $\theta, h \models P_1 \rightarrow* P_2 \Leftrightarrow (\forall h'. h' \# h \wedge \theta, h' \models P_1 \Rightarrow \theta, (h' \uplus h) \models P_2)$ where $h' \# h$ denotes that the heaps are disjoint. Practically, magic wands are helpful to reason about “the rest” of a structure. For example, iterating over a linked list often requires the introduction of the list-segment predicate `1seg`, presented in Ex. 15, in order to specify the beginning of the list that has already been visited. Instead, the list segment `1seg(x, y, vs)` can be replaced by the magic wand `list(y, vs') →* list(x, vs ++ vs')`, meaning that the total list can be recovered by combining this resource with the rest of the list.

To add support for magic wands, we extend the assertion language with magic wand assertions of the form $p(\vec{E}_1; \vec{E}_2) \rightarrow* q(\vec{E}_3; \vec{E}_4)$, where p and q are user-defined predicates. We chose this syntax as to syntactically capture the in-parameters and out-parameters for each side of the operator. Such a magic wand assertion forms a simple assertion with \vec{E}_1 , \vec{E}_2 and \vec{E}_3 as in-parameters, and \vec{E}_4 as out-parameters. To explain the division of in-parameters and out-parameters, we give a summary of the underlying algorithm for performing consumption of magic wands as implemented in Gillian. The algorithm is originally from Viper [23, 5]:⁵

To consume a magic wand assertion $p(\vec{E}_1; \vec{E}_2) \rightarrow* q(\vec{E}_3; \vec{E}_4)$ from state σ :

1. Create a state σ_p by producing the definition of $p(\vec{E}_1; \vec{E}_2)$ in the empty state.
2. For each simple assertion Q in the definition of $q(\vec{E}_3; \vec{E}_4)$:
 - a. Try consuming Q in σ_p , if it succeeds continue to the next simple assertion;
 - b. If it fails, try consuming Q in σ instead, if it succeeds, continue to the next simple assertion;
 - c. If both fail, abort the consumption.

Step 1 produces the left-hand side of wand, which requires knowing all its parameters. Therefore, all parameters in \vec{E}_1 and \vec{E}_2 must be in-parameters of the wand assertion. Then, step 2 consumes the right-hand side of the wand, which requires knowing all its in-parameters, but learns its out-parameters in the process. Therefore, \vec{E}_3 are in-parameters of the wand assertion and \vec{E}_4 are out-parameters.

To exemplify, say $p_1(x; y) = x \mapsto y$ and $q_1(x; y, z) = x \mapsto y \star y \mapsto z$. Further, say we know $x = 1$ and $y = 2$ and are in a state with a heap $\{2 \mapsto 3\}$. The heap satisfies the wand assertion $\exists z. p_1(x; y) \rightarrow* q_1(x; y, z)$. Indeed, by the above algorithm, the assertion can be consumed starting from the given heap, learning that $z = 3$ in the process.

⁵ For simplicity of presentation, the algorithm presented here assumes the absence of disjunction in the definitions of p and q .

6 Scalability and Performance

We now discuss the scalability and performance of MPs. We base our discussion on the MP implementation in Gillian, specifically, our discussion builds on the largest case study carried out in Gillian, i.e., the verification of C and JavaScript implementations of the deserialisation module of the AWS Encryption SDK message header [13]. First, we report MP-related scale and performance data for the AWS case study. Second, we report on a new MP-based optimisation we have implemented in Gillian for this paper, which allows for the creation of *aggregate matching plans* (AMPs). We show that this optimisation improves the total verification time of the AWS case study.

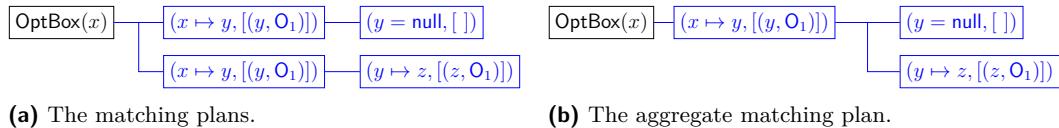
AWS case study. To measure the scale and performance of MPs, we have instrumented Gillian with data-and-performance counters and re-run the verification of the code from the AWS case study. From this experiment, we have found the cost of building MPs to be negligible compared to the total verification time. For the C/Javascript implementation of the AWS case study, building all MPs takes a total of 0.35s/0.096s, constituting 0.16%/0.25% of the total verification time. Over that time, MPs are built for 1073/378 assertions that consist of 41/28 simple assertions on average and 156/272 assertions at most. The creation of a single MP takes 0.33ms/0.26ms on average and 2.5ms/6.5ms at most. Note that MPs do not affect the verification time beyond the time it takes to create them; this is because MPs are separated from the consumption phase: the consumptions that take place during verification would be the same if the input assertions had instead been manually adopted (e.g., as illustrated in the discussion on VeriFast in §2).

Aggregate matching plans (AMPs). We discuss and evaluate *aggregate matching plans* (AMPs), a new performance optimisation we have implemented in Gillian for this paper. To illustrate AMPs, recall that an MP for a predicate is a list of MPs for the disjuncts of the body of the predicate (Def. 11), and that each disjunct is treated independently. AMPs identify and leverage simple assertions that are shared between disjuncts and represent this sharing within a tree structure.

To better understand how AMPs work, consider the following predicate:

$$\text{OptBox}(x) \triangleq (\exists y. x \mapsto y * y = \text{null}) \vee (\exists y, z. x \mapsto y * y \mapsto z)$$

and the MPs and AMP for this predicate in Fig. 3.



■ **Figure 3** Matching plans and aggregate matching plan for $\text{OptBox}(x)$.

Without AMPs, Gillian would create 2 MPs, one per disjunct of the predicate, which both have the same first step ($x \mapsto y, [(y, O_1)]$). However, when folding a predicate, Gillian tries to consume each disjunct of the predicate body in order until one succeeds to completion. Without AMPs, when the definition that could be folded was the second one, the first step would be consumed twice in the same symbolic state, duplicating the work. In contrast, using AMPs it is consumed only once, factoring out such duplicated work.

In our implementation of AMPs in Gillian, MPs are built for each disjunct of a predicate, and then aggregated into a single AMP. Before building the individual MPs, simple assertions within a single disjunct are sorted using a simple sort algorithm, maximising the chance of the existence of a shared root. A similar process is also performed for function specifications, as each function can have multiple specifications in Gillian.

Our evaluation of this new optimisation shows that utilising AMPs instead of lists of MPs in large verification projects leads to substantial performance improvements. For the C implementation of the AWS case study, AMPs made the total verification time drop from 240s to 211s, that is, a speedup of 12%. AMPs are especially effective when assertions are obtained from and/or augmented by a compilation process which often adds the same contextual information, such as type information, to all cases (which is the case for the Gillian assertion compiler for C).

7 Related Work

We place matching plans in the context of previous work on automated frame inference. Specifically, we compare matching plans with the approaches of three modern SL-adjacent and SMT-based semi-automated verification tools: VeriFast [7], Viper [16], and CN [20].

VeriFast. VeriFast [7], whose approach is closest to our work, is a verification tool for C and Java. It is based on consumers and producers, and its assertion language is the traditional SL assertion language. Given the similarities between VeriFast and our work, in particular the shared assertion language, we expect it would be straightforward to adapt our work on MPs for VeriFast. Currently, the approach of VeriFast offers less automation than MPs, as it leaves the responsibility of constructing MPs to the tool user, who has to provide the MP implicitly when providing assertions, e.g., as part of predicate definitions. To illustrate, consider again the singly-linked list predicate from our running example, now in VeriFast’s syntax for C:

```
struct node { int entry; struct node* next; };

predicate list(struct node* x, list<int> vs) =
  x == NULL
  ? vs == nil
  : malloc_block_node(x) && x->entry |-> ?v &&
    x->next |-> ?x' && list(x', ?vs') && vs == cons(v, vs');
```

Note that this list predicate is defined using the ternary conditional operator rather than disjunction, and that existentially quantified variables are annotated with a question mark at first use. In VeriFast, simple assertions are consumed in the order given by the tool user: e.g., in the non-`NULL` case of the list predicate, `malloc_block_node(x)` is consumed before `x->entry |-> ?v`, which in turn is consumed before `x->next |-> ?x'`, and so on. This means that if the user does not arrange the simple assertions appropriately, the verification will fail even though there might exist an MP. Further, VeriFast offers less automation than MPs w.r.t. learning variables: it can only learn a variable if that variable is the single occupant of an *out* or the left-hand side of an equality: for example, assuming `x` is known, VeriFast can learn `y` from `y == x` but not from `x == y` or `x == y + 1`.

Another difference between our approach and that of VeriFast is that in our approach *ins* and *outs* are checked at definition time whereas in VeriFast they are checked at use time, leading to less local/precise error reporting. For example, if we tried to fold a list in VeriFast

using `close list(_)`, where `_` denotes that VeriFast should infer the argument, we would get the error message “Unbound variable ‘`x`’”, referring to the `x` variable in the `list` predicate definition, instead of an error saying that it is not possible to infer an *in* of a predicate.⁶

Viper. Viper [16] is a platform for building verification tools. It has been instantiated, among other languages, to Java and Rust. It is based on consumers and producers, but also on an alternative assertion language known as implicit dynamic frame theory (IDF) [24], which combines SL with dynamic frame theory [10].⁷ Tool users familiar with SL but not IDF must therefore learn IDF before they can start using Viper. This difference also means that both consumption and learning *outs* from *ins* look different than in our setting, making a detailed comparison complex. We illustrate this using our linked-list running example, now in Viper’s IDF syntax:

```
field entry : Int
field next : Ref

predicate list(this : Ref) {
    acc(this.entry) && acc(this.next) &&
    (this.next != null ==> list(this.next))
}

function elems(this : Ref) : Seq[Int]
requires list(this) {
    unfolding list(this) in
    this.next == null ? Seq(this.entry)
                      : Seq(this.entry) ++ elems(this.next)
}
```

The above `list` predicate captures the shape, but not the contents, of lists. The predicate is expressed using `acc`, a construct called accessibility predicate, closely resembling the cell assertions in SL. The contents of lists are specified using a heap-dependent function `elems`. Such functions, as their name suggests, are functions over the heap of the current symbolic state. In the setting of accessibility predicates and heap-dependent functions, the *ins* look similar to *ins* in our setting, but the *outs* become the return values of heap-dependent functions. Assertions must be self-framing, in the sense that assertions must ensure accessibility to at least the locations they read. Self-framedness is checked in a left-to-right manner in Viper, meaning that the assertion `acc(x.f) && 0 < x.f` is considered self-framing, whereas `0 < x.f && acc(x.f)` is not. That is, like VeriFast, Viper is sensitive to the order of simple assertions. Quantifiers, e.g., over array indices, are more prominent in IDF than in SL, and variables, quantified or otherwise, that cannot be inferred are instantiated by giving the underlying SMT solver trigger hints [14], in the style of, e.g., Boogie [12].

CN. CN [20] is a verification tool for C, and is designed for, what its authors call, “predictable proof automation”. One means employed towards this goal is that CN is based on a new tool-specific assertion language, which uses variable scoping to ensure that *outs* can always be learnt. In other words, the limitations of CN’s learning algorithm are reflected directly in

⁶ VeriFast has support for checking “preciseness of predicates”, which allows for definition-time checking of their *ins* and *outs*. However, this feature does not affect the error reporting at use sites of predicates, i.e., errors remain nonlocal. The rules are the same as for the run-time check and are described by inference rules by Jacobs et al. [7] and by prose text by Jacobs et al. [8].

⁷ Parkinson and Summers [19] establish a formal connection between SL and IDF, and Jost and Summers [9] (partially) extend the result to include predicates as well.

syntax of assertions, ensuring that tool users do not accidentally fall out of the plannable subset of the assertion language. To exemplify, in CN syntax, the singly-linked list predicate for the same `node` data type as in the above discussion on VeriFast becomes:⁸

```
predicate { list<integer> l } List (pointer p) {
    if (p == NULL) {
        return { l = nil<integer> };
    } else {
        let Head = Owned<struct node>(p);
        let Tail = List(Head.value.next);
        return { l = cons(Head.value.entry, Tail.l) };
    }
}
```

Since new variables, including *outs*, must be the output of functions, they are necessarily learnable. The downside of this approach is that tool users have to learn a new specification language.

Another feature aimed at predictable proof automation is that CN targets a decidable SMT fragment, disallowing, e.g., nonlinear arithmetic in SMT queries. Instead, these must be handled manually by tool users, by proving lemmas in, e.g., Coq, and then manually applying them in CN. This trade-off is not CN-specific and could also be done in an MP-based approach. Similarly, manual fallbacks for complex quantifiers are required as well.

Other related work. Many other SL and SL-adjacent verification tools share similarities with the work presented here, all the way back from Smallfoot [2, 3], the very first such tool. Important differences between our work and Smallfoot include that Smallfoot is not SMT-based and that its frame inference procedure is more akin to proof search than the approach presented here, as is the case for its most well-known descendant Infer [4]. Another important approach to semi-automating SL is embedding one’s verification tool inside an interactive theorem prover (ITP). A recent example of this approach is RefinedC [22]. Such tools do not reduce the verification problem to a series of SMT queries but instead to a series of proof obligations that tool users must then discharge within the ITP by the usual means available, including various proof automation machinery.

There are also important connections to be highlighted between the work presented here and logic programming. The above-mentioned work on RefinedC [22] highlights this connection, as the tool is implemented in the “separation logic programming language” Lithium (which, in turn, is implemented in Coq), which is introduced in the same paper. Diaframe [15] is based on similar ideas. Nguyen et al. [17] highlight the connection between what we call *ins* and *outs* and argument modes in logic programming. Lastly, some logic programming languages contain features that address some of the problems of the traditional left-to-right evaluation order of logic programming, such as constraint logic programming and co-routining (e.g., `dif/2`) (cf. the recent survey by Körner et al. [11]).

Finally, an earlier version of MPs, dubbed unification plans (UPs), was briefly outlined by Fragoso Santos et al. [6] in the context of the JavaScript analysis tool JaVerT, the forefather of the Gillian platform. UPs featured a more limited form of learning than our MPs and were constructed purely syntactically: *ins* and *outs* were computed independently of a knowledge base, which meant that, for example, while it was possible to learn b from $b = a$ or $\text{list}(a; b)$ knowing a , it was not possible to learn c from $a = b + c$ or learn list lengths. No paper has covered UPs in the same depth we have covered MPs in this paper.

⁸ The definition is taken from the CN paper, where the authors add: “Note that CN does not currently support logical functions on lists; this example is for illustration only.”

References

- 1 Roberto Baldoni, Emilio Coppa, Daniele Cono D’elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys*, 51(3), 2018. doi:10.1145/3182657.
- 2 J. Berdine, C. Calcagno, and P. W. O’Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *International Conference on Formal Methods for Components and Objects*, 2005. doi:10.1007/11804192_6.
- 3 J. Berdine, C. Calcagno, and P. W. O’Hearn. Symbolic execution with separation logic. In *Asian Conference on Programming Languages and Systems*, 2005. doi:10.1007/11575467_5.
- 4 Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of C programs. In *NASA Formal Methods Symposium*, 2011. doi:10.1007/978-3-642-20398-5_33.
- 5 Thibault Dardinier, Gaurav Parthasarathy, Noé Weeks, Peter Müller, and Alexander J. Summers. Sound automation of magic wands. In *Computer Aided Verification*, 2022. doi:10.1007/978-3-031-13188-2_7.
- 6 José Fragoso Santos, Petar Maksimović, Gabriela Sampaio, and Philippa Gardner. JaVerT 2.0: Compositional symbolic execution for JavaScript. *Proceedings of the ACM on Programming Languages*, 3(POPL), 2019. doi:10.1145/3290379.
- 7 Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods Symposium*, 2011. doi:10.1007/978-3-642-20398-5_4.
- 8 Bart Jacobs, Jan Smans, and Frank Piessens. *The VeriFast Program Verifier: A Tutorial*, 2017. doi:10.5281/ZENODO.1068185.
- 9 Daniel Jost and Alexander J. Summers. An automatic encoding from VeriFast predicates into implicit dynamic frames. In *Verified Software: Theories, Tools, Experiments*, 2014. doi:10.1007/978-3-642-54108-7_11.
- 10 Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *Symposium on Formal Methods*, 2006. doi:10.1007/11813040_19.
- 11 Philipp Körner, Michael Leuschel, João Barbosa, Vítor Santos Costa, Verónica Dahl, Manuel V. Hermenegildo, Jose F. Morales, Jan Wielemaker, Daniel Diaz, Salvador Abreu, and Giovanni Ciatto. Fifty years of Prolog and beyond. *Theory and Practice of Logic Programming*, 22(6), 2022. doi:10.1017/S1471068422000102.
- 12 K. Rustan M. Leino. This is Boogie 2, 2008. URL: <https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/>.
- 13 Petar Maksimović, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. Gillian, part II: Real-world verification for JavaScript and C. In *Computer Aided Verification*, 2021. doi:10.1007/978-3-030-81688-9_38.
- 14 Michał Moskal. Programming with triggers. In *Workshop on Satisfiability Modulo Theories*, 2009. doi:10.1145/1670412.1670416.
- 15 Ike Mulder, Robbert Krebbers, and Herman Geuvers. Diaframe: Automated verification of fine-grained concurrent programs in Iris. In *Conference on Programming Language Design and Implementation*, 2022. doi:10.1145/3519939.3523432.
- 16 Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *Verification, Model Checking, and Abstract Interpretation*, 2016. doi:10.1007/978-3-662-49122-5_2.
- 17 Huu Hai Nguyen, Viktor Kuncak, and Wei-Ngan Chin. Runtime checking for separation logic. In *Verification, Model Checking, and Abstract Interpretation*, 2008. doi:10.1007/978-3-540-78163-9_19.
- 18 Peter W. O’Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Computer Science Logic*, 2001. doi:10.1007/3-540-44802-0_1.

26:20 Matching Plans for Frame Inference in Compositional Reasoning

- 19 Matthew J. Parkinson and Alexander J. Summers. The relationship between separation logic and implicit dynamic frames. In *European Symposium on Programming*, 2011. doi: [10.1007/978-3-642-19718-5_23](https://doi.org/10.1007/978-3-642-19718-5_23).
- 20 Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. CN: Verifying systems C code with separation-logic refinement types. *Proceedings of the ACM on Programming Languages*, 7(POPL), 2023. doi: [10.1145/3571194](https://doi.org/10.1145/3571194).
- 21 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, 2002. doi: [10.1109/LICS.2002.1029817](https://doi.org/10.1109/LICS.2002.1029817).
- 22 Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. RefinedC: Automating the foundational verification of C code with refined ownership types. In *International Conference on Programming Language Design and Implementation*, 2021. doi: [10.1145/3453483.3454036](https://doi.org/10.1145/3453483.3454036).
- 23 Malte Schwerhoff and Alexander J. Summers. Lightweight support for magic wands in an automatic verifier. In *European Conference on Object-Oriented Programming*, 2015. doi: [10.4230/LIPIcs.ECOOP.2015.614](https://doi.org/10.4230/LIPIcs.ECOOP.2015.614).
- 24 Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *European Conference on Object-Oriented Programming*, 2009. doi: [10.1007/978-3-642-03013-0_8](https://doi.org/10.1007/978-3-642-03013-0_8).

The Fault in Our Stars

Designing Reproducible Large-scale Code Analysis Experiments

Petr Maj

Czech Technical University, Prague, Czech Republic

Stefanie Muroya

Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria

Konrad Siek

Czech Technical University, Prague, Czech Republic

Luca Di Grazia

Università della Svizzera italiana (USI), Lugano, Switzerland

Jan Vitek

Charles University, Prague, Czech Republic

Northeastern University, Boston, MA, USA

Abstract

Large-scale software repositories are a source of insights for software engineering. They offer an unmatched window into the software development process at scale. Their sheer number and size holds the promise of broadly applicable results. At the same time, that very size presents practical challenges for scaling tools and algorithms to millions of projects. A reasonable approach is to limit studies to representative samples of the population of interest. Broadly applicable conclusions can then be obtained by generalizing to the entire population. The contribution of this paper is a standardized experimental design methodology for choosing the inputs of studies working with large-scale repositories. We advocate for a methodology that clearly lays out what the population of interest is, how to sample it, and that fosters reproducibility. Along the way, we discourage researchers from using extrinsic attributes of projects such as stars, that measure some unclear notion of popularity.

2012 ACM Subject Classification Software and its engineering

Keywords and phrases software, mining code repositories, source code analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.27

Funding This work was supported by the Czech Ministry of Education, Youth and Sports under program ERC-CZ, grant agreement LL2325, BigCode (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000421). NSF grants CCF-1910850, CNS-1925644, and CCF-2139612, as well as the GACR EXPRO grant 23-07580X.

Acknowledgements We would like to thank Digital Ocean for their involuntary contribution of computational resources during the early data gathering phase of our research. We acknowledge the reviewers of ICSE'22, and thank the reviewers of ECOOP'23 for their encouragements and for sticking around until 2024.

1 Introduction

And so it begins...

count the number of stars associated with each repository. The number of stars relate to how many people are interested in that project. Thus, we assume that stars indicate the popularity of a project. We select the top 50 projects in each language

Sentences like these appear in the methodology sections of software engineering papers, with sometimes, little more in terms of experimental design. This paper aims to convince readers that using extrinsic features of projects, such as popularity, may limit applicability of results of the studies relying on them. Instead one should select projects based on their intrinsic features and spell out expectations as well as threats to validity.

Empirical software engineering studies are experiments performed on a corpus of software to validate some hypotheses. For instance, one could take projects written in various languages and attempt to show that some language feature has an impact on the quality of the code written using it. The value of large-scale corpus study often does not lie in what we learn about the projects that were analyzed, but rather in what these can teach us about the larger population. There is limited value in, say, finding out that a new Java language feature is beneficial in a handful cases if we cannot generalize that result to a broader portion of the Java ecosystem.

Yet, many papers in the field do not articulate how broadly applicable their results are expected to be. Even the simple question of how the projects that were analyzed were selected is not clear. While large-scale code repositories, such as GitHub, are a boon to the software engineering community, their sheer size requires care. We argue that better experimental design will strengthen research done in the field.

Consider Table 1 which has a meta-study of three years of the *Mining Software Repositories* conference. Forty-one papers relied on subsets of GitHub. Out of those, five papers lacked sufficient information about their dataset to determine how they selected their inputs, twenty-one used GitHub stars to obtain a subset of projects, ten used simple combinations of attribute thresholds and only five relied on random sampling over the entire population.

■ **Table 1** Experimental design in MSR 2019, 2020 and 2021.

papers	class	description	# of projects
5	Unknown	Unknown or proprietary	1–35K
21	Stars	Filter projects using stars	5–2M
10	Other	Other filter for projects	7–290K
5	Random	Filter and sample randomly	6–51K

What is the right choice here? None of the papers analyzed the entire ecosystem as that would mean tens to hundreds of millions of projects. The question thus becomes how to sample the population of projects hosted on GitHub. In this paper, we criticize the use of GitHub stars as they are appealing and popular, yet also dangerous. But, really, our point generalizes to any extrinsic attributes of a project. So, again, what is the right choice? The answer is, of course, that it depends. The rest of this paper attempts to shed some light on how to make a reasoned choice of inputs.

First, let us return to stars and ponder why they play such an important role in our experimental methodology? We believe expectations and pragmatics are the explanation. Community standards are largely set by the papers we publish. The literature codifies expectations for authors of the next batch of papers. These expectations slowly evolve in response to reviewer attitudes. So, we use stars because our peers do. And just as importantly for the pragmatic reason that GitHub does not provide an index of projects, nor does it allow to query over intrinsic attributes of code. Finding inputs is thus hobbled by limitations of our tools. Stars play a double role. They are a queryable index of projects as GitHub does provide an interface to obtain them. They also come with an expectation that starred projects enjoy some notion of quality [8]. This paper will show that stars do not necessarily correlate with quality and that they introduce reproduction barriers.

We propose a methodology for designing reproducible software engineering experiments over large-scale repositories with the explicit goal of improving the generalizability of our results. The methodology is in line with evolving community standards [23] but specific to large-scale code analysis. We propose to follow the following protocol when designing a new experiment:

1. **Population Hypothesis:** Give a brief description of the population of interest the research should generalize to; it may be narrow such as “programs written by students learning JavaScript” or as broad as “commercial code”.
2. **Frame Oracle:** Give a procedure for deciding if a project belongs to the population of interest. The procedure should be efficiently computable over intrinsic attributes of a project. An oracle could, say, return projects with a single JavaScript file created by a user with no previous commits.
3. **Sampling Strategy:** Describe a strategy for selecting a subset of the entire population. Ideally, specified algorithmically. An example is random sampling without replacement from a known seed.
4. **Validity:** Give an argument as to how the oracle and the sampling strategy are valid means to obtain representative samples. This can be a discussion of how to check result quality, such as manual inspection of samples written by beginners, and threats to validity.
5. **Reproduction Artifacts:** Publish an artifact that reproduces exactly the reported results, and supports changes to either the input or the details of the experiment.

Reproducibility has nuances. Our emphasis is on providing support for the following three use cases: *Repetitions* which run the reproduction artifact to obtain bit-for-bit equal results. This is the most stringent use case and often requires a reproduction artifact that bundles code and inputs. *Reanalysis* alters either the method or its input, it requires an executable artifact and a method for acquiring new inputs. Finally, *reproductions* are independent implementations that require the paper to have an unambiguous description of all experimental details.¹ Supporting reproducibility can be greatly simplified with appropriate tooling. Our work builds on the CodeDJ infrastructure (codedj-prg.github.io). Our contributions are:

1. **A dataset** of 2Mio+ projects with intrinsic attributes precomputed.
2. **A characterization of stars** as a means to select inputs for code analysis experiments.
3. **A methodology** that can be readily adopted to improve reproducibility.
4. **A reproduction** that highlights challenges to generalization due to project selection.

Our community has been moving towards broader adoption of the practice of artifact evaluation [11]. While artifacts are clearly helpful as they make papers providing them easier to reproduce, the selection of inputs is often hardwired and not considered part of the reproduction. The impact of our proposal, if adopted, would be to encourage authors of large-scale code studies to consider the collection of the inputs to their work to be part of the experiment and thus make it easy to change the way inputs are selected.

Road map. The structure of the paper is as follows.

- Section 2 begins with a short overview of the state with respect to methodologies for project selection and tooling to support it.
- Section 3 takes four practical examples, papers published at the Mining Software Repositories conference, and attempts to couch their experimental design in the terms introduced above. These example suggest that authors are not always clear about their intent and strategy. While looking at these papers we found a number of practical impediments to reproducibility.

¹ The terminology comes from [24] and was used by SIGPLAN artifact evaluation committees.

- Section 4 describes characteristics of the projects hosted on Github and argues that stars cannot yield a representative sample of developed projects.
- Section 5 outlines our proposal for how to design large-scale program analysis experiments.
- Section 6 follows our guidelines and attempts to repeat the studies of Section 3 while perturbing the experimental inputs.
- Section 7 is responsive to reviewers of this paper and their request to reproduce an experiment from a paper with a verified artifact.
- Section 8 concludes and gives some parting thoughts. This paper improves on the state of the art in that it argues for a structured experimental design that relies on tooling for input selection.

2 Related work

We review relevant advice, warnings and the state of tooling.

2.1 Community standards

A push towards reproducibility is underway. The standards framework of Ralph et al. [23] includes a section on experimental design and specifically on sampling. This is further explored by Baltes and Ralph [1]. They argue that software engineering faces a generalizability crisis. In their meta-analysis of 120 papers, they find that convenience sampling² is widely used to select projects to analyze from a large population. Convenience sampling rarely leads to representative samples, and – without a careful study of potential sources of bias – can lead to conclusions that do not generalize. They explain this state of affairs by a fundamental challenge: the lack of appropriate sampling frames to access elements of the population of interest. Earlier work by Nagappan et al. [19] already attempted to address this problem by defining the notion of sample coverage as a way to assess the quality of the data used as input to an experiment. Even closer to our paper is the study by Cosentino et al. [4] which reported that out of 93 large corpus papers, 63 papers failed to provide replication datasets. Most papers did not use random samples and omitted mentions of any limitations.

2.2 Mining repositories

GitHub is a popular data source. Warnings about its perils go back to the work of Kalliamvakou et al. [10] which highlighted “noise” among hosted projects. In particular, they point out, tiny and inactive projects dominate the platform. Lopes et al. [13] poured oil on that fire, showing that up to 95% of the files containing code in some language ecosystems were copies of one another and filtering by stars reduces the proportion of duplicates without eliminating them. One way researchers have strived to find signal in GitHub’s sea of noise is to use stars. But what do stars mean? We would like them to be correlated with quality code, code worth analyzing. Borges and Valente [2] conducted a user survey that found the most common reasons for starring a project was to show appreciation (e.g. *starred this repository because it looks nice*) and bookmark it (e.g. *starred it because I wanted to try it later*). They also warn against promotional campaigns to drive up ratings. Popularity of projects was studied by Han et al. [8], they found that while users believe stars are a measure of a project’s popularity, intrinsic attributes such as branches, open issues and

² The Wikipedia definition of convenience sampling is a type of non-probability sampling that involves the sample being drawn from that part of the population that is close to hand.

contributors are better predictors. Expanding on that result, Munaiah et al. [18] propose classifier for *engineered projects*, which they define as projects that leverage sound software engineering principles. They show that the classifier outperforms stars. Pickerill et al. [22] further improved classification with an approach based on time-series clustering.

2.3 Tools for miners

A number of infrastructures have been developed to assist researchers in the field. The most ubiquitous was, the now defunct, GHTorrent [6]. The project offered a continuously updated database of metadata about public projects that was a valuable building block for other tools. Boa is complementary as it lets users write sophisticated queries over source code [5]. CodeDJ is a newer infrastructure that supports queries over both meta-data and file contents and is language agnostic [15]. Recent works address performance issues of querying at scale [14, 17]. Of these, only CodeDJ ensures reproducible queries.

3 State of practice

How do people design experiments for large-scale code studies? This section gives some examples that we believe to be representative which we will revisit later when we attempt to reproduce the results with different inputs. For each paper, we provide a brief summary of the scientific claims made by the authors. Then, we attempt, with our best understanding of the work, to reverse engineer a version of the protocol laid out in the introduction. We, thus, give an account of each paper’s population hypothesis, a description of the frame oracle, sampling strategy, validation and reproduction artifacts. We conclude the section with some observations general reproduction issues that show up in these papers.

3.1 MSR 2020: What is software

“Software” has an intuitive definition, namely code, but there is more. The paper by Pfeiffer [21] classifies the content of repositories in categories such as code, data and documentation. They, then, observe that software is more than just code. Documentation is an integral constituent of software, and software without data is often correlated with libraries, and finally that software without code is rare, but exists. The paper answers the question “*what are the constituents of software and how are they distributed?*” The paper argues that existing definitions of the term are non-descriptive, inconclusive and even contradictory.

Population Hypothesis: Implicitly, the population is all inclusive.

Frame Oracle: Given the lack of details, we assume all projects on GitHub belong.

Sampling Strategy: the authors carry out convenience sampling by choosing popular repositories. Stating “*by popularity we mean the starred criteria with which GitHub users express liking similar to likes in social networks.*” Most-starred projects in 25 languages were acquired by executing one query by language, saying that “*without language qualifier, the API returns only 1,020 repositories in total, which we decided is not enough for our study.*”

Validity: No discussion of relevant issues or threats.

Reproducibility Artifacts: A listing of files and repositories is provided along with the code of the classifier and a notebook. Repository contents were not preserved.

3.2 MSR 2020: Method chaining

In an object-oriented language, a *method chain* occurs when the result of a method invocation is the receiver of a subsequent invocation. In Java, chains manifest as sequences of calls connected by dots. Nakamura et al. [20] analyze trends in usage of method chains and conclude that they increase over a period of eight years.

Population Hypothesis: Java projects developed “*by real-world programmers*.” The authors state that they “*did not apply any filter to the collected repositories. This supports the generalizability of our results*.” The authors also consider generalization beyond Java, saying “*our results are more likely to be applied to a language that does not provide such a construct (e.g. PHP and JavaScript). The empirical study of this hypothesis is future work*.”

Frame Oracle: Implicitly, all Java projects hosted on GitHub.

Sampling Strategy: The authors use convenience sampling, taking 2,814 projects that appeared at least once in the list of the 1K most-starred projects in November 2019. Projects were deduplicated and filtered for syntactically invalid files.

Validity: No discussion of relevant issues.

Reproducibility Artifacts: Project metadata and computed chain lengths are available. Communication with the authors reveals that their reproduction package is not available.

3.3 MSR 2019: Style analyzer

Each software project seems to develop its own formatting conventions. Markotsev et al. [16] demonstrate that an unsupervised learning algorithm can automate project-specific code formatting. They reproduce styles with a high degree of precision for a set of repositories.

Population Hypothesis: The authors speak of “*real projects*” and their artifact support JavaScript, so we assume an expectation that the projects “developed” in a sense similar to [18].

Frame Oracle: All developed JavaScript projects hosted on GitHub.

Sampling Strategy: Convenience sampling yielded 19 JavaScript projects with high star counts.

Validity: Authors manually inspected projects in the selection.

Reproducibility Artifacts: A GitHub repository containing the tool and a file with project URLs along with their head and base commits is provided. Contents of repositories are not included. Run scripts did not run out of the box.

3.4 MSR 2020: Code smells

Code smells are programming idioms correlated with correctness or maintenance issues. Jebnoun et al. [9] contrast code smells in projects related to deep learning and general purpose software. Their claim is that for large and small projects there is a statistical difference in the occurrence of code smells, whereas medium sized projects are indistinguishable.

Population Hypothesis: The paper focuses on two populations: projects related to deep learning, and general purpose software. For pragmatic reasons, they focus on Python as it is popular for machine learning.

Frame Oracle: Python projects with keywords indicating machine learning, discarding tutorials. Furthermore, the authors “*also carefully select popular and mature DL projects from them by employing maturity and popularity metrics (e.g., issue count, commit count, contributor count, fork count, stars)*.”

Sampling Strategy: A staged strategy was employed. The authors relied on judgment sampling to manually select 59 deep learning projects. For general purpose projects, they used a top-starred list of 106 Python projects from [3] and randomly sampled 59 projects. Projects were further clustered into small ($\leq 4,000$), medium, and large ($\geq 15,000$) based on size.

Validity: No issues were discussed.

Reproducibility Artifacts: A listing of the 59 deep learning projects is provided.

3.5 Summary and discussion

The papers we have reviewed do not explicitly talk about any of the four points in our protocol, in all cases we had to reverse engineer (or guess) some of them. This suggests that our proposal would improve the generalizability of the research.

While the mentioned research projects were done with care, there were challenges reproducing them out of the box. Common sources of reproducibility failures that occur in the papers we have reviewed are:

- *Missing descriptions:* Failure to specify either one of: population hypothesis, frame oracle or sampling strategy. Reproduction is fraught with perils and an apple-to-apple comparison between papers is difficult. This affects [21, 20, 16, 9] as their descriptions are open to interpretation.
- *Missing projects:* Even with a list of URLs, the corresponding projects may vanish at any time (e.g., deleted or made private). Reproductions are partial at best, we have seen a project disappear while being downloaded. This affects [21, 20, 16, 9].
- *Fading stars:* Stars are volatile. [20] observed close to 3,000 projects in the top 1K during a period of two months. Without a history of star attribution and a timestamp, reconstructing the star listings is not possible. Stars volatility also caused problems for [9].
- *Shifting contents:* The contents of a project change with new commits. To reconstruct the data, ids of the last observed commit must be specified. Even that is not foolproof as Git histories can be updated destructively. This affects [21, 20, 9].
- *Language attribution:* Projects contain code in many languages. For reproduction attribution must be specified. While delegating to, e.g. GitHub, is reasonable, one should be aware that GitHub has changed their attribution algorithm several times. Double counting a project is sometimes valid. This affects [21, 20, 16].
- *Deterministic replay:* Non-determinism must be limited. Random sampling seeds should be specified. This affects [16].

4 Mapping the GitHub landscape

The meta-study of Table 1 highlights the dominant position of GitHub as a data source in large-scale code analysis studies. The size of GitHub is such that it is necessary to resort to sampling to yield manageable datasets. As shown in the previous section, authors often look for some notion of “developed” projects, that is, they want projects that contains code of some quality.

We claimed that convenience sampling using stars as a proxy for various other characteristics of “real-world” software is flawed. While this may sound plausible to some readers, it should be backed up with data. Given the size of GitHub, this section uses sampling to answer the following questions: *Are starred projects a representative sample of all projects?*

and *Are starred projects a representative sample of developed projects?* where what it means for a project to be developed is purposefully left open as there is no agreement on a precise definition of the term.

Since the later parts of this paper require Java, Python and JavaScript, we acquire samples of these three ecosystems. We use CodeDJ to do this. It is an open source project that allows users to create a dedicated input project database and ensure reproducibility of queries.

We used random sampling over the entire GHTorrent dataset to select which projects to acquire in each of the languages of interest. The number of downloaded projects is somewhat arbitrary as it is based on available hardware during the acquisition phase. The datastore has 1,111,950 Java projects, 216,602 Python projects and 1,259,856 JavaScript projects. To give an idea of the scale, our Java dataset accounts for 20% of all non-forked GitHub Java projects. To get a manageable size, we down-sample further, randomly selecting 1Mio Java and JavaScript projects, and 200K Python projects.

4.1 Attributes

With CodeDJ, it is easy to write queries that compute project attributes. For this paper, we calculate five attributes that highlight the differences between projects:

- **C-index:** A developer handle has a C-index of n if that developer was party to at least n commits to n projects (i.e. n^2 commits). The C-index of a project is the highest such number across developers. This measures developer expertise.
- **Age:** The age of a project is the number of days separating the first commit and the most recent commit. This correlates with the maturity of a project.
- **Devs:** The count of unique developer handles in the git logs; includes both the author of a code change and the committer of that change. Devs approximates the size of a team, as some individuals may have more than one handle this is an upper bound.
- **Locs:** The total number of lines in files that are recognized as code, in any language, and appear in the head of the default branch.
- **Versions:** A version is implicitly created for each commit touching a file, be that for creation, deletion or update. This counts versions in the entire project's history including branches. Versions measure the activity in a project.

While we make no claims that these attributes suffice to fully describe a software project, we have found them to be an effective summary in many interesting dimensions.

4.2 Stars v. All

What do these attributes tell us about the overall population and about starred projects? If starred projects were representative of the entire population, they would share the same statistical distribution.

Fig. 1 is a histogram of each attribute; the x-axis is log scaled values in the unit of each attribute, the y-axis is the proportion of projects normalized for the maximum height. Grey denotes the whole population, red and blue denote the 1K most starred projects written in Java and Python respectively. The black, red and blue bars denote the median of their respective populations.

Consider the grey bars for the whole population, when comparing Java and Python, we see the same general shapes. The C-index is low, with a median of 2 for Java and Python. This means that half of the projects hosted on GitHub, have developers who have made at most two commits. The median age of Java projects is 7 days, while Python projects trend

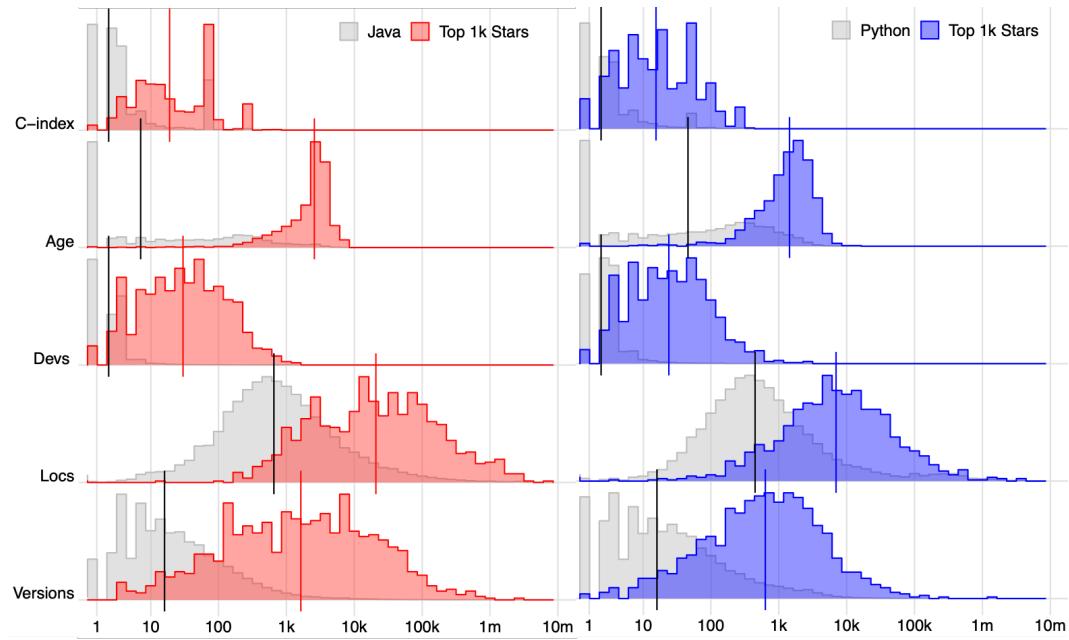


Figure 1 Comparing datasets.

slightly older, 46 days. The median number of developers for both languages is 2. As for median lines of code, Java project are slightly larger than Python, 655 compared to 448. The median number of commits (versions) is 16 for both languages. Overall, this confirms that most projects are small, short-lived and created by relative newcomers.

The top 1K starred projects have a very different make up. Visually it is clear from the fact that every distribution is shifted right. Starred projects are larger, older, with more experienced developers. While there are slight differences between languages, the overall picture is consistent.

Consider for instance, the C-index and age attributes. While many starred projects are team efforts, a significant number of projects have few contributors. Their C-index is high, with median of 19 for Java and 15.5 for Python, suggesting that experienced developers tend to contribute to popular repositories. The median age projects is more surprising with 2,581 and 1,440 days. Manual inspections suggest that many starred projects are indeed long lived but also have been inactive for years. Projects rarely “loose” stars, so if a project gets to the top there is a chance it will stay there long past its useful lifetime.

The answer to our first question is clearly negative. Starred projects are not representative of the overall population. This is not necessarily a bad thing, as folklore suggests that most of GitHub is uninteresting. Perhaps it is the case that starred projects are more “interesting.”

4.3 Stars v. Developed

Researchers often look for *engineered* [18, 22] or *developed* projects – informally, projects created with some care – alas there is little agreement on a precise definition.

Slightly easier, perhaps, is to settle on what we do not want, the projects that are uninteresting, one that are clearly of little value for any reasonable research question. Moreover, one could hope that the complement of uninteresting projects are the projects researchers look for. Let us define a project as *uninteresting* if it has less than 100 lines of

code, fewer than 7 days old, and fewer than 10 commits. When this definition is used to filter projects, this rather low bar manages to eliminate 71% of Java and 55% of Python projects. For the purpose of this discussion we term the remaining projects are developed or interesting. It would be nifty if stars were a proxy for filtering out uninteresting projects.

Fig. 2 shows the distribution of the whole population in Grey (in the same way as in Fig. 1), the interesting projects in Black and the top 1K starred projects in Red. Clearly, the shape of the Black and Red distributions do not align suggesting that stars do not represent interesting projects.

Manual inspection of the starred project highlights their main issue – stars are extrinsic properties without a direct connection to any attributes of a project. Unlike our computed attributes, stars grow monotonically. Their meaning is unclear as users award them for various reasons including humor and shock value. Some projects earned stars because of a joke not fit for this audience (e.g. github.com/dickrnn/dickrnn.github.io), or have dared users to star junk (github.com/gaopu/java). Stars do not measure quality or usefulness of repositories.

To further illustrate the limitation of stars as a filter, we take, for each attribute, the 20 projects with the lowest score for that attribute. Table 2 shows a manual classification of these projects. None of these projects is useful: externals lack histories, widgets are small and biased by their application domain, babies are too small to yield much insights, and the remaining ones only have code snippets.

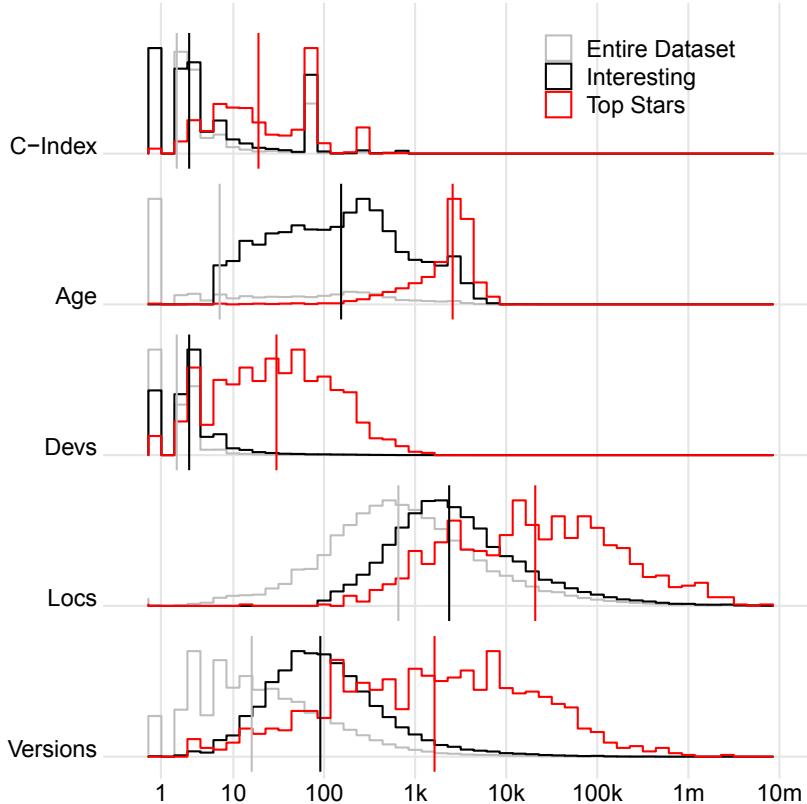


Figure 2 Comparing developed and starred projects.

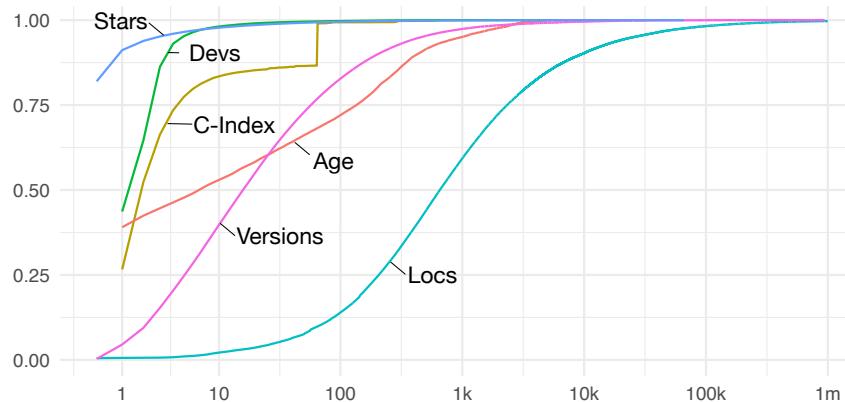
■ **Table 2** Categorizing 200 starred projects.

Category	Java	Python	Description
Externals	9%	5%	Infrequent synchronization with another repository.
Widgets	43%	0%	Tiny projects with little activity, popular UI widgets or plugins.
Docs	4%	15%	Interview questions, course materials, games, knitting patterns.
Tutorials	17%	9%	Educational materials, tutorials and example applications.
Babies	16%	32%	Valid but extremely small projects with little activity.
Artifacts	0%	21%	Research artifacts developed elsewhere and deposited for sharing.
Deprecates	1%	5%	Deprecated projects, no code on the main branch.

The answer to our second question is also negative. Starred projects are not representative of the interesting ones. To summarize what we learned about stars, they capture extrinsic characteristics of GitHub projects and are at best an indirect and noisy proxy for a robust frame oracle.

4.4 How to select projects?

What to use for project selection if not stars? We argue that selection must be based on intrinsic features – measurable attributes of a project’s contents. While one may use machine learning [18, 22] to build classifiers, in this paper we will use our five computed attributes (we leave machine learning as an interesting area of future work).



■ **Figure 3** Cumulative Density Functions.

Fig. 3 is the cumulative density function of the various attributes for Java (the shapes of the curves for Python are similar). The interpretation of each line is what percentage of the dataset is filtered for a particular attribute value. Project selection can be performed by a combination of attributes with cutoffs. We do not argue for a particular formula; researchers must make their own choices in this respect.

For instance, if one were to use 10 days of age as a cutoff, then 52% of the dataset would be filtered out. Whereas picking a 10 star cutoff, filters out 98% of projects.³

³ The discontinuity of C-Index at 65 is odd. After manual investigation, we found that there is a single developer with that C-index, it turns out that the “developer” is a bot doing automated updates.

4.5 Validity of our dataset

We noticed an oddity around project ages in our dataset. Experience with GitHub trained us to expect the unexpected. Our investigation started with a plot of creation dates.

Fig. 4 shows the log scaled counts of new projects over time. While there is a steady progression in the count of projects created each year, we see a significant drop in 2015 and a plateau until 2019.

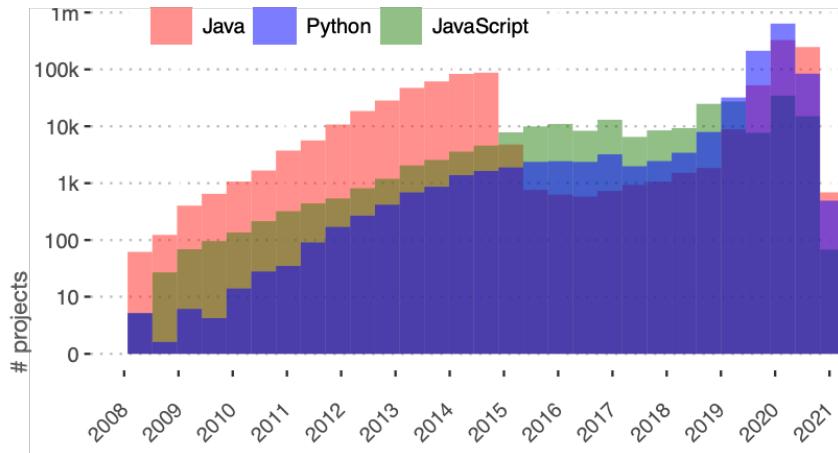


Figure 4 Creation date.

We reviewed our pipeline to no avail. We use GHTorrent to acquire all available URLs. Then, we randomly sample projects from that list. We validated both acquisition and sampling. This leaves us with two hypotheses. First is a consistent flaw in the CodeDJ downloader causing some projects to fail to download. 17% URLs obtained from GHTorrent point to dead projects, but there is no apparent bias. Second some projects could be missing in GHTorrent.

Another issue showed up on inspection, JavaScript project ages are significantly higher than those of other languages. We found that GitHub timestamps are frequently inconsistent. Why should JavaScript be more affected? Until an explanation can be found, we removed JavaScript from the overall comparison and use JavaScript projects in the reproduction with extreme care.

5 Reproducible large-scale analysis experiment design

This paper proposes that researchers conducting experiments over large-scale software repositories follow a specific experimental design methodology to ensure their work can be reproduced and increase chances that their results generalize as expected. While the mechanics of reproducibility of the actual experiment itself vary, the setup of the experiment is a common problem. The proposed methodology has five steps, we encourage researchers to document each of these steps explicitly.

5.1 Population hypothesis

Formulating a population hypothesis lets researchers stake a claim about the applicability of their work. This represents the population to which the result of an experiment should generalize to. The statement of that hypothesis can be brief and appeal to intuition, the other parts of the description flesh out the details.

Ideally, we would like our results to be as broadly applicable as possible, but pragmatically designing experiments that back up overly broad claims is difficult. Some populations of interest are difficult to sample, for instance “commercial software” is a relatively simple and unambiguous description but one that we typically cannot sample from as most of the commercial software is not in the public domain. Other populations can be difficult to identify. Imagine a study of the challenges linked to retraining imperative programmers to use functional idioms. Finding code written by such developers can be done manually but is difficult to automate. It is often easier to describe a population by intrinsic features of projects such as the language used to write the code or some estimate of the size of the project.

5.2 Frame oracle

A frame oracle is a, possibly noisy, deterministic algorithm for deciding if a project belongs to the population of interest. The oracle is our best approximation of the population of interest. An executable and reproducible oracle allows to compare different papers with the same selection. The description of the oracle should specify the data source along with any information required to acquire projects. The procedure for evaluating a project should be clear and based on intrinsic attributes. A paper should at least have a short description of the oracle, full details should be given in the reproduction artifact.

5.3 Sampling strategy

The literature has an abundant advice on sampling (see e.g. [12]). Briefly, a sampling strategy picks the type of sampling (probabilistic or non-probabilistic) and describes the steps used to obtain a sample. The sampling implementation is expected to be found in the reproduction artifact.

Many works use convenience sampling as it is simpler, cheaper and less time consuming. A better alternative is some form of probabilistic sampling as it is more likely to yield a representative sample. Probabilistic sampling can be staged if the structure of the population is more complex. The simplest approach is random sampling where each element has the same chance of being picked. We often have to resort to stratified sampling when the population is divided into subgroups of different sizes. Typically we sample without replacement as we do not want to pick the same project multiple times.

5.4 Validity

The validity section argues, when there are reasons for doubt, why using the frame oracle and the sample strategy results in representative samples of the population of interest. This section should address potential sources of bias and attempts by the authors to control for them. This section also should address any foreseen challenges to reproducibility and offer means to mitigate them.

5.5 Reproducibility artifacts

Finally, we advocate to link the paper to a reproduction artifact that contains code and data to support experimental repeatability and reanalysis.

Section 3 listed issues with reproducibility. In some cases, the authors did not give a precise description of the steps needed for reproduction. Following our proposed methodology along with a reproduction artifact will greatly help.

The second category of issues are more pragmatic, it is difficult to repeat the analysis of a paper because some aspect of the data used is not available. We suggest that research infrastructures should support this task.

An example of an infrastructure is CodeDJ which is both a continuously updated datastore and a database that can be queried by a DSL. We adopted it for our work and illustrate how it helps with reproducibility. The implementation of a frame oracle and sampling strategy can be combined into a single expression. Fig. 5 shows a query which starts by filtering out projects containing fewer than 80% JavaScript code, then it uses pre-computed attributes Locs, Age and Devs to filter further. The last stage of filtering involves computing an attribute on the fly, here we sum up the commits in the project, before performing random sampling.

```
database.projects().filter(|p| {
    p.language_composition().map_or(false, |langs| {
        langs.into_iter().any(|(lang, rate)| { lang == JavaScript && rate >= 80 })
    })
})
.filter_by(AtLeast(Locs, 5000))
.filter_by(AtLeast(Age, Duration::from_months(12)))
.filter_by(AtLeast(Devs, 2))
.filter_by(AtLeast(Count(Commits), 100))
.sample(Random(30, SEED))
```

Figure 5 Project selection with CodeDJ.

CodeDJ is split between a persistent datastore in which every data item is timestamped, and an ephemeral database used to service queries. A reproducible query is a Rust crate archived in a git repository associated to the datastore. Running the query produces a *receipt* which is the hash of a *commit* automatically added to the archive repository. The receipt can be used to share the query (exactly as executed) and its results (exactly as produced). It can be used to retrieve the Rust crate and re-execute the code. Code re-execution is helped by the fact that queries are deterministic and the crate contains a list of all dependencies, a timestamp, and all random seeds. When a query with an embedded date is executed, CodeDJ accesses the exact state of the datastore at the specified date. Since CodeDJ stores the contents of files, entire experiments can be fully reproduced.

6 Reproductions

We illustrate the use of the proposed methodology by revisiting the papers we discussed in section 3. For each paper, we attempt reproduction where we vary the input. The original papers used stars in their selection, we will explore different inputs based on intrinsic attributes.

If the results of the reproduction match the original results, then it suggests that stars were an appropriate filter. If the reproduction departs, this suggests that there may be need to conduct further experiments to be confident in the results.

For each paper, we picked a subset of the scientific claims to fit the reproduction in the available space. We use our proposed methodology to describe the details of the reproduction.

One may wonder how we selected the paper to reproduce. Our criteria were (1) papers that used automated techniques to analyze properties of the code contained in Github projects, (2) their population of interest was a large subset of Github, (3) a working artifact could be located, and (4) the input could be changed with ease. We did not cherry-pick as even positive results would be interesting. Our choice was limited by the fact that many

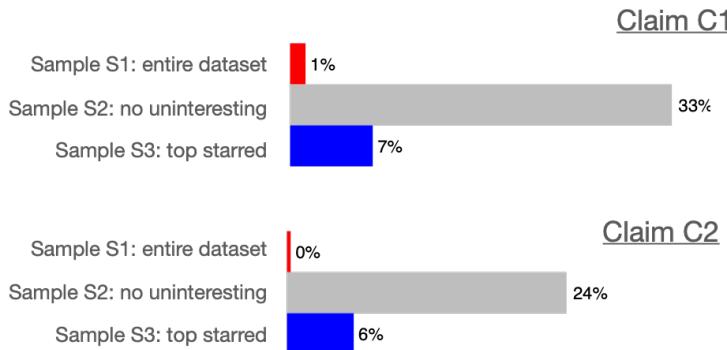


Figure 6 Proportion of projects without code, data or documentation.

papers either did not have artifacts or that they were not working anymore. Furthermore, some papers had hardwired their selection of projects by, for example, preprocessing the input data and omitting to include the tooling to repeat that processing. Given more time, more works could be reproduced.

6.1 Reproduction: What is software

This reproduction aims to validate two findings of [21]: (C1) 4% of repositories do not contain code, data and documentation; (C2) 2% of repositories do not contain documentation.

Population Hypothesis: The universe of software projects.

Frame Oracle: To understand the impact of project selection we consider two oracles. O1 accepts any project hosted on GitHub. O2 removes uninteresting projects (as defined above).

Sampling Strategy: We report on three samples. S0 is a convenience sample of starred following [21]. S1 and S2 are random samples without replacement from O1 and O2 respectively, stratified by language and deduplicated.

Validity: Our reproduction differs in the number of languages (3 v. 25) and by categorizing files based on the file path alone. We tested stability of our results with multiple samples of varying sizes and manually inspected the produced labels.

Reproduction Artifact: Our artifact contains a CodeDJ receipt for this query.

6.1.1 Results

Fig. 6 shows results for claims C1 and C2. Compare the percentages between S0 (original) and S1 (target population). Statistical analysis is not required to see that the difference is significant. The sample S2 (without uninteresting projects) is there to illustrate the impact of slightly more developed population, but even these are still quite different.

Would the results agree if we included more languages? The three languages we downloaded account for most of GitHub, it is conceivable that other languages could affect results, but that would just push the generalizability issue somewhere else as the claims would become language-specific.

6.2 Reproduction: Method chaining

Nakamura et al. [20] claim that 50% of projects in 2018 had method chains longer than 7 while in 2010 that number was 42%. They state that “chains of length 8 are unlikely to be composed by programmers who tend to avoid method chaining, this result is another supportive evidence for the widespread use of method chaining.”

Population Hypothesis: The universe of real-world Java programs.

Frame Oracle: We accept any Java project hosted on GitHub and delegate to Github for language attribution.

Sampling Strategy: Stratified sampling to randomly select projects with commits in 2010 and 2018.

Validity: To reproduce the original results, we performed stratified sampling to get top starred projects active in the target years. The authors used a different sample of top stars. The original paper had different sample sizes for each year, but those are not specified. We fix the sample size to 250. The authors could not locate the code of their chain detector, so we use our own implementation.

Reproduction Artifact: Our artifact contains a CodeDJ receipt for this query.

6.2.1 Results

Fig. 7 shows the difference in proportion of projects at various chain lengths. The solid line uses stars, colors represent different random samples. For instance, if we pick chains of length 8, the number used by [20], the difference is a 13% increase in the number of projects between 2011 and 2018. The differences for our random samples are -2%, 0.6% and 0.7%. In other words, the samples from this particular population do not seem to show the effect expected by the authors. We surmise that some notion of developed project may show more favorable results, but without more guidance in the population hypothesis it is hard to guess which to pick.

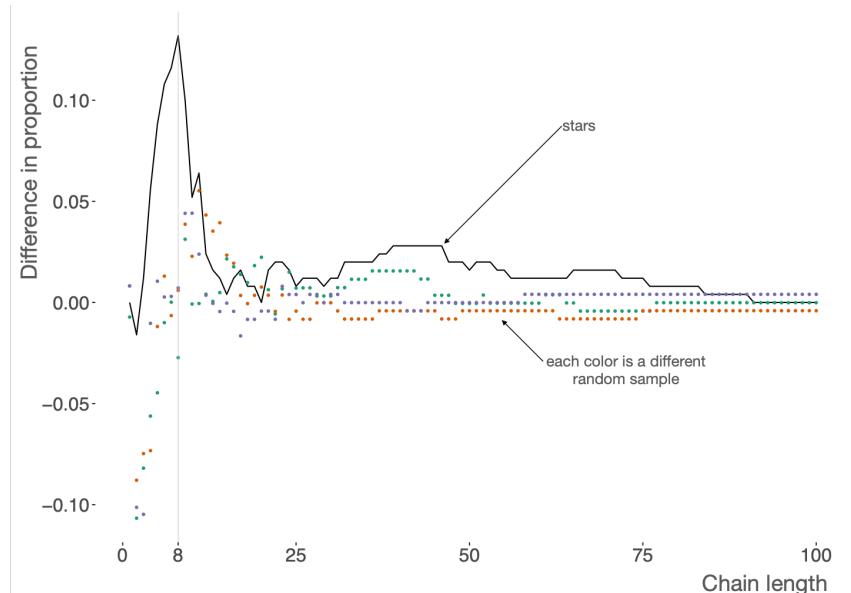


Figure 7 Difference in chain lengths.

6.3 Reproduction: Style analyzer

Markovtsev et al. [16] builds a model of the style of a repository and apply this model on a held-out part of that repository to produce corrections. Their experiment uses 19 top-starred JavaScript project to gauge the precision with which the tool flags formatting discrepancies and the relationship between this precision and the size of the project. They report a precision of 94% (average, weighed by project size) and better overall performance for large projects and projects with better style guidelines.

Population Hypothesis: Developed JavaScript projects.

Frame Oracle: JavaScript projects with at least 80% JavaScript code, $\text{Loc} \geq 5000$, $\text{Age} \geq 12 * 31$ and $\text{Devs} \geq 2$.

Sampling Strategy: We randomly select 10 sets of 30 projects. This is more projects than the original sample to account for errors in processing. After processing is finished, following the original paper, we randomly select 19 projects out of the pool of successfully processed projects.

Validity: Given that the author's artifact lacks a configuration, we used the default one. This increases project size, as compared to the published numbers, by 38% per project (up to a maximum of 154%) and causes precision to diverge by 2.2% on average, and up to 7.9%.

The tool failed to process 4 projects: `freecodecamp` and `atom` due to errors in unicode processing, `express` due to a programming bug, and `30-seconds-of-code` due to bad file identification. Three of the missing projects were located close to the median in terms of precision, prediction rate, and project size in the original paper, while `axios` was in the lower quartile for sample count.

Style analyzer analyzes each project at two points in its history specified by a base commit and a head commit. The base commit is a point in the past which the tool checks out to learn the project's formatting style. The head commit is a more recent point used to evaluate the model and calculate precision. The original paper provides head and base commits for each project in their experiment, but does not specify the method of selecting these commits. We pick the current head of the default branch as the head commit. For base commit we pick one that lies at an offset equal to 10% of the number of all commits in the default branch from the head commit. This retrieves different commits than the original paper, which causes a 3% median change in precision (up to 17%– `telescope`) and a median project size increase of 76%, and up to 311% (`reveal.js`).

Reproduction Artifact: Datasets, receipts from submitted queries, style analyzer's reports and scripts for the entire experimental pipeline are included in the artifact.

6.3.1 Results

We recreate a plot of the effect of the number of items in the training set on precision from the original paper in Fig. 8. The training set consists of snippets created around tokens/AST nodes relevant to formatting (whitespace, indentation, quotes, zero-length gaps). We plot the selection from the original paper along three selections from our interesting project frames. In addition, we plot the distributions of precision in each selection in

In Fig. 9, we compare the precision scores in each sample with the selection used in the original paper using a Mann-Whitney U test to show which samples performed statistically differently from the original. The scatter plots show a different grouping of results from the original paper. The groupings in the scatter plot visibly differ between selections. The distribution comparison shows that our selections generate significantly smaller training

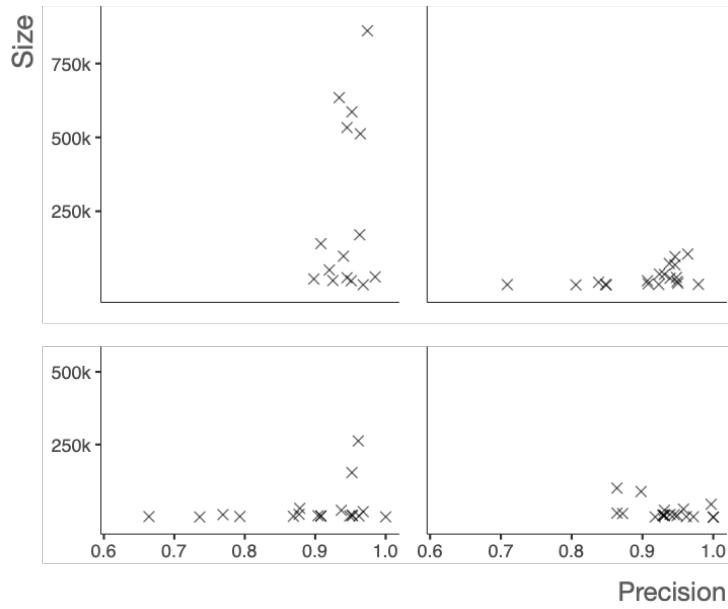


Figure 8 Relationship between label groups and precision.

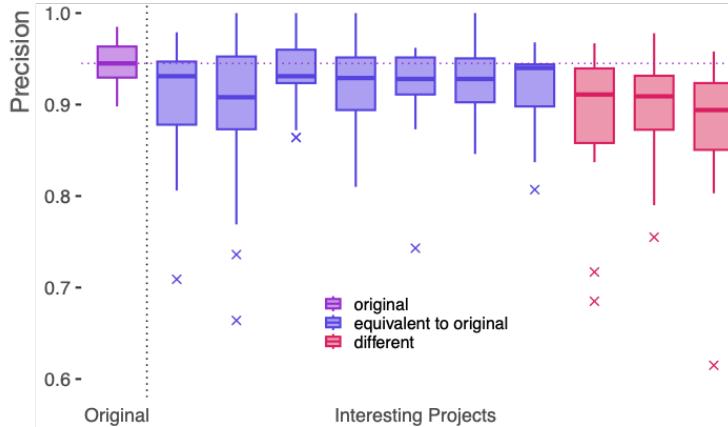


Figure 9 Comparing label group count and precision.

sets in all cases and yield lower precision. In addition, 3 out of the 10 interesting project selections produced significantly lower precision, with the remainder producing a statistically equivalent distribution.

Overall, we see our selections yielding precision between 0.9 and 0.95 (the paper sets a precision of 0.95 as a benchmark for success). We also do not see a clear relationship between the number of label groups and precision, such as the one the authors note in the original paper.

6.4 Reproduction: Code smells

We seek to validate the claim of [9] that for large and small projects there is a statistical difference in the occurrence of code smells between machine learning and most popular Python repositories, whereas medium sized projects are indistinguishable.

Population Hypothesis: Mature Python projects in all application domains including machine learning.

Frame Oracle: Projects with $C\text{-Index} \geq 5$, or $\text{Age} \geq 180$, or $\text{Locs} \geq 10,000$, or $\text{Versions} \geq 100$.

Sampling Strategy: The deep learning projects were provided by the authors. Out of 59 projects, 57 were still accessible on August 2nd 2021. At download time there were 6 small, 13 medium, and 38 large deep learning projects. For the reproduction of the original results, we used a staged strategy, first convenience sampling the top starred Python projects and amongst those used stratified sampling to select 57 projects with a similar distribution of sizes. To generalize the results we used quota sampling to match the size distribution.

Validity: Our reproduction uses the Locs reported by CodeDJ. The date the authors downloaded the repositories is unknown. We use the content of the main branch of each repository as of April 1st, 2020. The authors say “*each of repositories is pre-processed and prepared for code smell detection*”, however details are missing. We used the default thresholds of their tool.

Reproduction Artifact: A CodeDJ receipt is included in our reproduction package along with code to run the experiment.

6.4.1 Results

Fig. 10 contrasts the distribution of code smells for deep learning projects, top starred projects, and three random samples. Computing the p-values with the non-parametric Mann-Whitney Wilcoxon shows that while we were able to reproduce the statistically significant results for the small projects, we disagree on the large most popular projects with the original. The disagreement is even greater in the random samples where no large projects and only one small project is statistically different. Generalizability of the results is thus questionable.

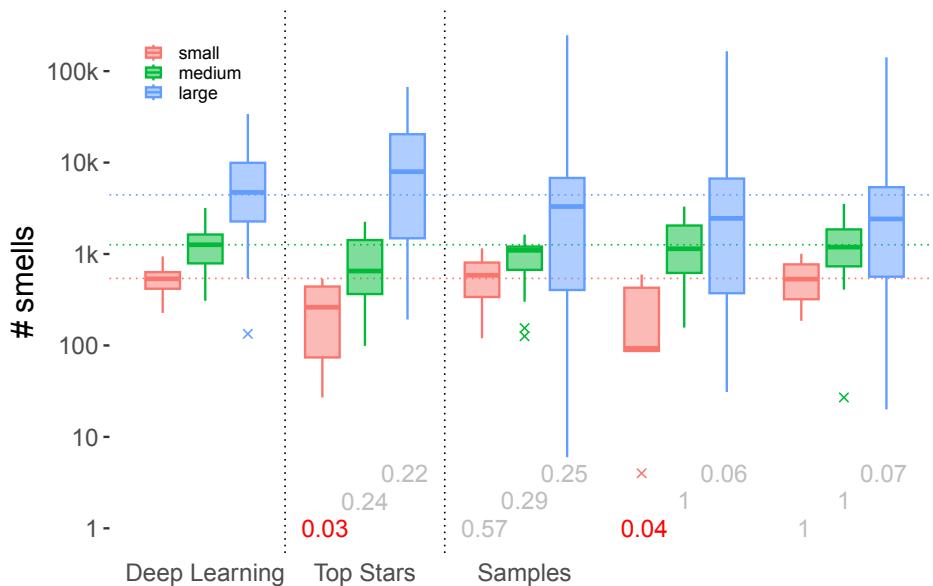


Figure 10 Comparing Smells. Numbers are p-Values indicating a significance of the difference from the deep learning projects. Statistically significant different p-Values (cutoff at 0.05) are shown in color, insignificant ones are in gray.

7 Collaborative Reproduction

We perform one last reproduction in which we obtain the assistance of the study’s authors to validate whether stars are a good input selection strategy. For this reproduction we selected a distinguished paper from the Foundations of Software Engineering conference (2020), “The Evolution of Type Annotations in Python” [7]. The paper has a reusable artifact for repetition of the original results. Our reproduction only required to change the list of GitHub URLs used as input in the analysis. The authors helpfully allowed us to run the rather computationally intensive workload on their machine.

The original study reported the following protocol for input selection. *“We group projects by creation date, considering projects created in the years 2010 to 2019, into ten groups. We sort each group by number of stars and select the top-1000 per group, which yields a total of 10,000 projects. The rationale for first grouping and then sampling is to avoid biasing our study toward projects created in a particular time frame, e.g., mostly old projects. Removing projects that we could not clone, e.g., because they became unavailable since the beginning of our study, the total number of analyzed repositories is 9,655.”*

The first research questions was related to the evolution in the number of type annotations in projects. The main insight from the work was that *“better developer training and automated techniques for adding type annotations are needed, as most code still remains unannotated, and they call for a better integration of gradual type checking into the development process.”*

For this reproduction, we discussed input selection criteria with the authors and arrived at the following formulation.

Population Hypothesis: Python projects in all application domains with earliest commit date in 2015⁴ as this was the year when early adoption of type annotations began.

Frame Oracle: Python projects whose life span is longer or equal to 7 days and that have over 100 lines of Python code. The study authors intended their work to be representative of most of the Python ecosystem, but closer inspection of some of the small projects suggested that they would introduce noise. The particular cutoffs were chosen heuristically.

Sampling Strategy: Projects were grouped by year active and, for each year, a random sample of 1200 projects was selected. The goal was to get close to a thousand usable projects for each year. As some of the projects in our database are no longer available, the sample size is increased heuristically.

Fig. 11 illustrates the reproduction results (and mirrors Fig. 2 in [7]). The plot on the left shows the number of type annotations found per thousand lines of code in the projects being looked at. The red line has the new data, the blue one is for the original study. The y-axis is logarithmic. The two lines start at zero in 2015. Both data sets tell a similar story: type annotations are gradually added to projects. The plot on the right shows the total number of annotations found each year in all projects. The y-axis is in thousands. In 2021, the original data had close to 800,000 annotation while the new data is under 250,000.

Both data sets are large. The original one contains 1,123,393 commits and the new data set 1,535,824 – suggesting that projects are slightly larger in the randomly selected data than in the most starred projects. In both cases, only a fraction of the repositories have types. In the old data 668 repositories are type-annotated, whereas in the new data 1,040 projects have at least one type. The fraction of commits that change a type annotation is small in both cases 5.5% in the original data and 2.1% in the new data.

⁴ 2015 is when type annotations were added to Python.

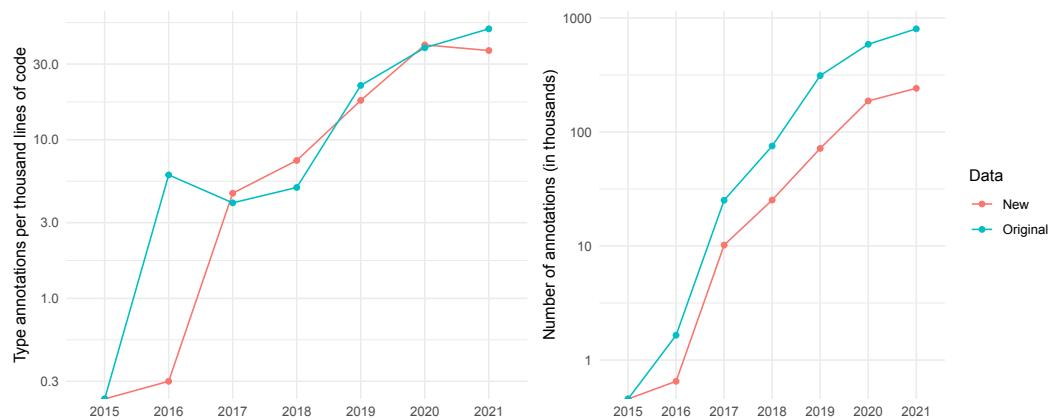


Figure 11 Types in Python.

Overall, the reproduction verifies and, even, strengthens the conclusion of the original paper. Five years after introduction of type annotations, their use remains rather limited. Having said this, it is true that actual values reported are different enough to be noticeable.

8 Conclusions

Sometimes, doing it wrong is so much easier than the alternative, that we convince ourselves that a little wrong can be right enough. Our paper is unusual. While it purports to contain a call to arms for better experimental practices, it is just as much a record of our own journey to that goal. What reads as criticism is really written in self-reflection. So, what can a researcher take away from this paper? There are three ideas we would like to leave the reader with.

Generalizability. The value of an experiment often lies as much in what it generalizes to, as in the experiment's outcome. We found that many researchers rely on GitHub stars to pick representative samples of software projects, yet starred projects tend to be larger in most dimensions than typical ones, also that they are more likely to be inactive, and that their ranking is not a measure of intrinsic qualities of the code. Hopefully, this paper is the last nail in that coffin. More generally, we advocate for the use of probabilistic sampling over populations defined by intrinsic attributes of software, and also for clear and standardized documentation of experimental design.

Reproducibility. The value of a scientific experiment also lies in our ability to reproduce it. Carrying out reproducible experiments over large-scale software repositories is hard. Especially when aiming to support the three reproduction modalities: repetition, as practiced in artifact evaluation, where an artifact is re-executed to obtain identical results; reanalysis, where the artifact or its input are modified; and independent reproduction, where the entire experiment is re-implemented from scratch. The first modality requires faithful replay and is best served if all data used is included with the artifact. The second, requires support for automatically acquiring new representative samples. The third needs an unambiguous description of all experimental steps. We advocate for reproductions artifacts that supports the first two modes, and a detailed description of the experiment for the last.

Tooling. Generalizability and reproducibility, while worthy goals, represent much work, and they are work that is orthogonal to the scientific goals of researchers. The only reasonable answer is to provide tooling that automates acquisition of representative samples and generation of reproduction artifacts. In this paper, we used CodeDJ and found it helpful as it let us specify queries over attributes of the code for many projects, while also supporting experimental repetition and reanalysis through historical queries. It has its limitations, we found execution times to be somewhat long and doubt it will scale to the whole of GitHub.

Our vision for a brighter future is one where the community agrees on standard tools and techniques for this kind of experiment, tools which automate the acquisition and packaging of input datasets and the re-execution of entire experiments.

References

- 1 S Baltes and P Ralph. Sampling in software engineering research: a critical review and guidelines. *Empir. Softw. Eng.*, 27(4):94, 2022. doi:10.1007/s10664-021-10072-8.
- 2 H Borges and M Túlio Valente. What's in a github star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 2018. doi:10.1016/j.jss.2018.09.016.
- 3 Z Chen et al. Understanding metric-based detectable smells in python software. *Information and Software Technology*, 2018. doi:10.1016/j.infsof.2017.09.011.
- 4 V Cosentino, J Izquierdo, and J Cabot. Findings from GitHub: Methods, datasets and limitations. In *Mining Software Repositories (MSR)*, 2016. doi:10.1145/2901739.2901776.
- 5 R Dyer, H Nguyen, H Rajan, and T Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Int. Conf. on Software Engineering (ICSE)*, 2013. doi:10.5555/2486788.2486844.
- 6 G Gousios and D Spinellis. GHTorrent: GitHub's data from a firehose. In *Mining Software Repositories (MSR)*, 2012. doi:10.1109/MSR.2012.6224294.
- 7 L Di Grazia and M Pradel. The evolution of type annotations in python: An empirical study. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2022. doi:10.1145/3540250.3549114.
- 8 J Han et al. Characterization and prediction of popular projects on GitHub. In *Computer Software and Applications Conf. (COMPSAC)*, 2019. doi:10.1109/COMPSAC.2019.00013.
- 9 H Jebnou et al. The scent of deep learning code. In *Mining Software Repositories (MSR)*, 2020. doi:10.1145/3379597.3387479.
- 10 E Kalliamvakou et al. The promises and perils of mining GitHub. In *Mining Software Repositories (MSR)*, 2014. doi:10.1145/2597073.2597074.
- 11 S Krishnamurthi and J Vitek. The real software crisis: repeatability as a core value. *Commun. ACM*, 58(3), 2015.
- 12 S Lohr. *Sampling: Design and Analysis*. Cengage Learning EMEA, 2010.
- 13 C Lopes et al. Déjà Vu: A map of code duplicates on GitHub. *Proc. ACM Program. Lang. (OOPSLA)*, 2017. doi:10.1145/3133908.
- 14 Y Ma et al. World of code: enabling a research workflow for mining and analyzing the universe of open source vcs data. *Empirical Softw. Eng.*, 2021. doi:10.1007/s10664-020-09905-9.
- 15 P Maj et al. CodeDJ: Reproducible queries over large-scale software repositories. In *European Conf. on Object-Oriented Programming (ECOOP)*, 2021. doi:10.1145/2658987.
- 16 V Markovtsev et al. Style-analyzer: fixing code style inconsistencies with interpretable unsupervised algorithms. In *Mining Software Repositories (MSR)*, 2019. doi:10.1109/MSR.2019.00073.
- 17 T Mattis, P Rein, and R Hirschfeld. Three trillion lines: Infrastructure for mining github in the classroom. In *Conf. on Art, Science & Eng. of Programming <Programming>*, 2020. doi:10.1145/3397537.3397551.

- 18 N Munaiah et al. Curating github for engineered software projects. *Empirical Software Engineering*, 2017. doi:10.1007/s10664-017-9512-6.
- 19 M Nagappan, T Zimmermann, and C Bird. Diversity in software engineering research. In *Foundations of Software Engineering (FSE)*, 2013. doi:10.1145/2491411.2491415.
- 20 T Nakamaru et al. An empirical study of method chaining in Java. In *Mining Software Repositories (MSR)*, 2020. doi:10.1145/3379597.3387441.
- 21 R Pfeiffer. What constitutes software? In *Mining Software Repositories (MSR)*, 2020. doi:10.1145/3379597.3387442.
- 22 P Pickerill et al. Phantom: curating github for engineered software projects using time-series clustering. *Empir Software Eng*, 2020. doi:10.1007/s10664-020-09825-8.
- 23 P Ralph. SIGSOFT empirical standards released. *Softw. Eng. Notes*, 46(1):19, 2021. doi:10.1145/3437479.3437483.
- 24 J Vitek and T Kalibera. R3: Repeatability, reproducibility and rigor. *SIGPLAN Not.*, 2012. doi:10.1145/2442776.2442781.

Static Basic Block Versioning

Olivier Melançon 

Université de Montréal, Canada

Marc Feeley 

Université de Montréal, Canada

Manuel Serrano 

Inria/UCA, Inria Sophia Méditerranée, Sophia Antipolis, France

Abstract

Basic Block Versioning (BBV) is a compilation technique for optimizing program execution. It consists in duplicating and specializing basic blocks of code according to the execution contexts of the blocks, up to a version limit. BBV has been used in Just-In-Time (JIT) compilers for reducing the dynamic type checks of dynamic languages. Our work revisits the BBV technique to adapt it to Ahead-of-Time (AOT) compilation. This Static BBV (SBBV) raises new challenges, most importantly *how to ensure the convergence of the algorithm* when the specializations of the basic blocks are not based on profiled variable values and *how to select the good specialization contexts*. SBBV opens new opportunities for more precise optimizations as the compiler can explore multiple versions and only keep those within the version limit that yield better generated code.

In this paper, we present the main SBBV algorithm and its use to optimize the dynamic type checks, array bound checks, and mixed-type arithmetic operators often found in dynamic languages. We have implemented SBBV in two AOT compilers for the Scheme programming language that we have used to evaluate the technique's effectiveness. On a suite of benchmarks, we have observed that even with a low limit of 2 versions, SBBV greatly reduces the number of dynamic type tests (by 54% and 62% on average) and accelerates the execution time (by about 10% on average). Previous work has needed a higher version limit to achieve a similar level of optimization. We also observe a small impact on compilation time and code size (a decrease in some cases).

2012 ACM Subject Classification Software and its engineering → Just-in-time compilers; Software and its engineering → Source code generation; Software and its engineering → Object oriented languages; Software and its engineering → Functional languages

Keywords and phrases Compiler, Ahead-of-Time Compilation, Optimization, Dynamic Languages

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.28

1 Introduction

Optimizing compilers perform various analyses to discover properties of the program that are preconditions for performing optimizations. In a Just-In-Time (JIT) compiler, the cost of these analyses and optimizations is a critical issue as the time they take becomes part of the program's execution time. The use of expensive analyses and optimizations incur a long *warm-up* where the first part of a program's execution is sluggish and the program may even terminate before it has reached an optimization steady state.

Basic Block Versioning (BBV) is an optimization approach that strikes a balance between the optimization cost and the speed of the generated code to achieve a fast warm-up time and reasonably good execution speed. BBV has been used in JIT compilers for dynamically typed programming languages; in research compilers for JavaScript [5, 6] and Scheme [27, 28], and it is now used successfully in production in the official Ruby implementation [7, 8].

BBV uses the program's Control Flow Graph (CFG) created by the compiler as a template for creating a specialized CFG. For this, BBV traverses the CFG starting at its entry point while keeping track of the *context* that contains program properties of relevance

for specialization, such as the type of the values contained in the live variables. Each basic block has a set of contexts. The information contained in this set is *conservative*: for each possible program state when that basic block is reached during a program execution, there must be at least one context consistent with that state. Due to its conservative nature, it is allowed to have unreachable contexts in the set. In principle, each basic block of the original CFG could be specialized to all contexts in its context set, including unreachable contexts. The specialized versions of a basic block may contain optimizations that are valid in the corresponding context, such as the elimination of type checks when the type of a value has been determined at an earlier point in the execution.

An important concern is that multiple specialized copies of each basic block may be created, leading to a larger amount of code (*bloat*) and a longer compile time, load time, and execution time (due to the reduced performance of the instruction cache, among other reasons). In theory the bloat can be exponential in the size of the program.

Previous works use the same approach to this issue: a cap is placed on the number of versions for each basic block (*i.e.*, the number of versions is no greater than N , typically a small number like 5 or 10). In a JIT compiler the versions of a basic block are generated as the program's execution advances and reaches a basic block with a new context. This variant of BBV is called *lazy* BBV. When a new context is encountered and this would cause the version limit to be reached, a version specialized to that context must not be created because it would prevent another specialization if one was needed later in the execution. Instead, a *fully generic* version that covers all possible contexts is created as the last version and is used whenever a new version would be needed. Lazy BBV is relatively simple to implement but it has some important limitations:

- **Lazy BBV is a greedy algorithm.** The versions that are generated before the generic version, which are the first ones encountered at execution time, may not be the versions that are part of hot code. For example, if some function is used both during the initialization phase and in the main part of the program, then the specializations will be focused on what happens in the initialization phase. This function may be hot code when called from the main part of the program in a new context and, because the version limit is reached, it will be using the generic version (likely the slowest of them all).
- **High specialization is hard to achieve reliably.** The precision of the versioning context, *i.e.*, the number and information content of the program properties it tracks, has a direct impact on how quickly the slow generic version is used. For example, a precise versioning context that tracks not only the type but the range of values of an integer loop iteration variable starting at 1 and incremented at each iteration, will be able to create specialized versions of the first few iterations of the loop body (one version for each specific value of the iteration variable below N). This will not work well for loops that have a large number of iterations, because the iterations N and above will be handled by a slow generic version of the loop body. On the other hand, a context of this precision will work very well for programs where the loops have fewer than N iterations because BBV will completely unroll these loops. The BBV implementer will have to choose a moderate precision of the versioning context to avoid using the generic version too quickly, and consequently this will miss optimizations in some cases, such as total loop unrolling.
- **It requires JIT compilation.** The nature of a JIT compiler makes it easy to ensure that versions for unreachable contexts are never created. Unfortunately, this entails a warm-up time at execution, and in some use cases JIT compilation is not an option. In [5], an *eager* variant of BBV suitable for an Ahead-of-Time (AOT) compiler was described and compared to lazy BBV. That implementation of eager BBV yielded comparatively

poor speed and bloat because specialization is not guided by the actual need of a program execution and parts of the CFG that are explored are not typically executed, such as error cases and out-of-line handlers. Consequently, the specialized versions created before the limit is reached are more likely to be irrelevant at improving execution speed.

In this paper we describe a new design for a BBV algorithm suitable for an AOT compiler that mitigates these limitations. Our algorithm also traverses the CFG to determine which contexts reach each basic block. The first main difference with previous work is the handling of the version limit. When a new context is encountered and this would cause the version limit to be exceeded, the algorithm heuristically chooses a pair of contexts reached for that basic block and replaces them by a *merged* context that is more conservative than the contexts in the pair (in other words, a more general context). The algorithm continues traversing the CFG until a fixed-point is reached, *i.e.*, no new versions need to be created. The use of context merging allows contexts to be very precise at first, and it is the algorithm that reduces the precision as needed to keep the number of versions within the limit. The second main difference with previous work is the refinement of the notion of types to integer intervals to allow the BBV optimization to remove integer arithmetic overflow checks and array indexing bound checks.

In the next section, we present our algorithm and discuss its termination. In Section 3 we extend the algorithm with more precise contexts. The implementation in two mature compilers is explained and evaluated in Section 4. Related work is given in Section 5.

2 The Static BBV Algorithm

In this section we present the *Static* BBV (SBBV) algorithm. We will start with an overview by illustrating the algorithm's behavior using the traditional `find` function that many dynamic and functional languages provide.

2.1 SBBV by Example

The `find` function takes a predicate and a list of values and it returns the first element of the list that satisfies the predicate or false if no such element is found. In the Scheme programming language [17], which we use throughout this paper, it can be defined as:

```

1 (define (find p x)          ;; p is the predicate and x is the list to search
2   (if (pair? x)            ;; is the list non-empty?
3     (if (p (car x))       ;; call predicate on the first element
4       (car x)              ;; return it if it satisfies the predicate
5       (find p (cdr x)))    ;; otherwise, continue searching the rest of the list
6     #f))                  ;; return false when no element in the list satisfies the predicate

```

The safety of this code is guaranteed by verifying the validity of the arguments of the primitive operations at run time. In this example, the primitive operations for which a type verification is needed are the `car` and `cdr` accessors (lines 3-5) and the function invocation of the predicate (line 3). In safe mode, a Scheme compiler adds the required dynamic checks to the code, making the possible points of failed safety checks explicit:

```

1 (define (find p x)
2   (if (pair? x)
3     (if ((if (procedure? p) p (fail)) (if (pair? x) (car x) (fail)))
4         (if (pair? x) (car x) (fail))
5         (find p (if (pair? x) (cdr x) (fail))))
6     #f))

```

Here we use calls to the `fail` function to indicate cases where execution cannot continue due to a failed verification. The `fail` function is special in that it never returns.

One might expect a smart compiler, such as one implementing *occurrence typing* [32], to discover that the type tests on x are redundant, but let us assume that no such optimization is applied. This is done for illustrative purpose and because one of our objectives is to show that SBBV subsumes other optimization techniques, such as occurrence typing.

The unoptimized CFG of `find` is displayed in Figure 1a. SBBV will produce an optimized version of that CFG with fewer dynamic checks. For that, it propagates the information about variables in order to produce *specialized* versions of the basic blocks. For instance, block #12 (Figure 1a) checks that the end of the list is not yet reached by testing if x is a pair. In the positive branch, that is the path starting at block #4, it is known to be a pair and no further type tests are needed to ensure the correct execution until an assignment to x occurs (*i.e.*, the other tests that x is a pair are redundant).

The CFG produced by SBBV is shown in Figure 1b. We observe that SBBV has isolated the first iteration of the loop of the `find` function and the other iterations are handled by the loop formed by the subgraph $\{ \#23, \#25, \#27, \#29, \#31, \#32 \}$. In that loop the type of p is never tested because it has been tested in the first iteration before entering the loop handling the other iterations, and the argument x is only tested once per iteration, which is a necessary part of the loop termination logic. We also observe that for this simple example, SBBV has produced an optimal CFG in the sense that in a dynamic context with no global knowledge about the variable types and the data structure types, it executes the minimum number of dynamic checks required to ensure a safe execution. In Section 4.2 we evaluate the number of type tests SBBV is able to remove on more realistic programs.

In the rest of this section, we present and explain the algorithm that transforms the graph of Figure 1a into that of Figure 1b.

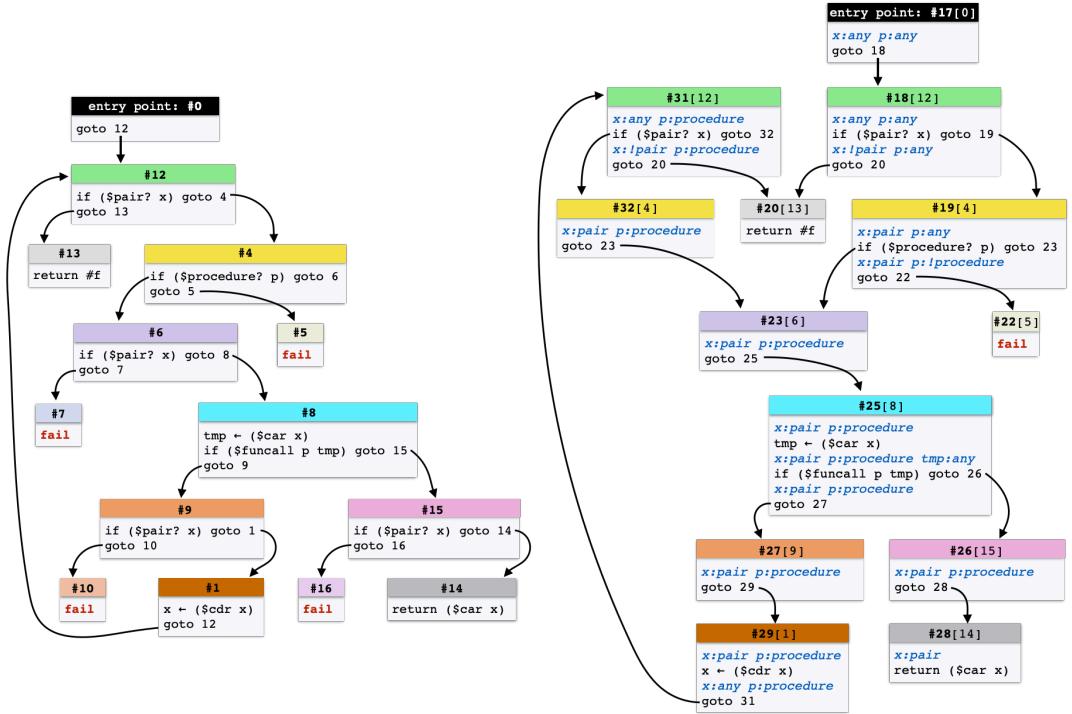
2.2 The Algorithm

SBBV specializes a CFG, which might either denote the whole program to be compiled or only a fragment of it. For instance, it can be decided to use SBBV for specializing each function in isolation or to specialize the whole program at a time. The main function of the algorithm (Algorithm 1) takes a basic block to specialize and the initial context used for that specialization as parameters. The data structures it uses are presented in Figure 2. A context is a mapping of variable names to value information. In this section these mappings associate variables to types. We will see in Section 3.2 that contexts can contain more precise type information. The algorithm specializes each instruction of the basic block and it recursively specializes the blocks that follow the basic block currently under specialization.

The algorithm performs a breadth-first traversal of the CFG. For that, it uses a work queue where it pushes the basic blocks and contexts that need further specializations. Specializing a block may cause the algorithm to specialize new blocks. For instance, when the algorithm scans the block #12 of Figure 1a, it discovers that in the positive branch the variable x is known to be a pair and then, it pushes onto the work queue the demand of a new specialization of the block #4 for the context $\{ x \mapsto \text{pair} \}$.

If the block extracted from the queue has not been merged (we explain in a moment what it means for a block to be merged), then it is specialized (line 10). This, in turn, can add new pending specializations to the queue. The algorithm proceeds until the queue is empty.

To ensure the convergence of the algorithm, it is enough to ensure that the function `BLOCKNEWVERSION` (Algorithm 2) pushes a new block onto the queue if and only if no such block has already been specialized for the requested context as the contexts contain a finite number of variable to value mappings. However, this convergence criteria is not enough in practice, the code size expansion of the specialization should also be controlled. This is the



(a) Original unspecialized CFG.

(b) Specialized CFG.

Figure 1 The original and specialized CFGs of the `find` function. Basic blocks have a numeric label for easy reference. In the specialized CFG, basic blocks have the same color as the original block they were specialized from (whose label is in square brackets). For instance, block #19 is a specialized version of block #4 of the original CFG. In the specialized CFG, the contexts of the specializations are displayed in blue. It can be observed that a given block can be specialized multiple times. For instance, block #12 has been specialized twice (blocks #18 and #31) and block #4 has been specialized twice (blocks #19 and #32). Blocks #7, #10, and #16 have no version in the specialized CFG because they do not have a reachable specialized context. Blocks #9 and #15 have been specialized to an unconditional jump because the `($pair? x)` test is known to be true.

context: ctx, ctx_0, \dots
■ types: a list of mappings of variable to type information
basic block: bb, bb_0, \dots
■ instrs: a list of instructions
■ versions: a list of specialized basic blocks
specialized basic block: bs, bs_0, \dots
■ bb: the corresponding basic block
■ merge: a specialized basic block into which it has been merged or false
■ ctx: a context

Figure 2 The data structures used in the SBBV algorithm.

28:6 Static Basic Block Versioning

Algorithm 1 Main algorithm.

```

1: function SBBV( $bb_0, ctx_0$ )           > specialize a CFG starting with the block  $bb_0$ 
2:    $wq \leftarrow empty\_queue$           > create a fresh queue used for this specialization
3:    $bs_0 \leftarrow \text{BLOCKNEWVERSION}(bb_0, ctx_0, wq)$  > push the request for specialization of  $bb_0$ 
4:   while  $\neg wq.\text{isempty}()$  do
5:      $bs \leftarrow wq.\text{pop}()$            > fetch the first bb of the queue
6:      $bb \leftarrow bs.\text{bb}$               > get the original unspecialized bb
7:     if  $|\{\forall b \in bb.\text{versions}, \neg b.\text{merge}\}| > \text{VERSION\_LIMIT}(bb)$  then
8:        $\text{BLOCKMERGESOME}(bb, wq)$       > too many specializations, merge
9:     if  $\neg bs.\text{merge}$  then
10:       $\text{BLOCKSPECIALIZE}(bs, wq)$     > specialize the block only if not already merged
11:   return  $bs_0$                   > return the specialization of the initial block

```

purpose of the test at line 7 of Algorithm 1. If the number of specialized versions of a single block exceeds a threshold, which is a parameter of the algorithm, some specializations of that block must be merged (see line 8).

The ancillary function BLOCKNEWVERSION is responsible for creating new blocks to be specialized. First, it checks if the requested block already exists, in which case it returns it. Otherwise, it creates a fresh version and pushes it onto the queue (line 9). Note that at this stage the instructions of the block are not scanned nor specialized. Pushing the block onto the queue is a mere request for specialization. It might be the case that at the moment where the block will be popped from the queue that this block has been merged into a less specialized version. This happens to prevent code size explosion.

Algorithm 2 Create or fetch a specialized version.

```

1: function BLOCKNEWVERSION( $bb, ctx, wq$ )
2:   if  $ctx \in bb.\text{versions}$  then
3:     return  $\text{BLOCKLIVE}(bb.\text{versions}[ctx])$       > return the already specialized block
4:   else
5:      $bs \leftarrow \text{new basic block}$            > create a fresh empty basic block
6:      $bb.\text{versions}[ctx] = bs$              > connect the new block and the parent block
7:      $bs.\text{bb} = bb$                       > initialize the new block
8:      $bs.\text{ctx} = ctx$ 
9:      $wq.\text{push}(bs)$                   > push it onto the queue for future specialization
10:    return  $bs$ 

```

The utility function BLOCKLIVE returns the first specialized version of a block that has not been merged into a more general version.

Algorithm 3 Follow a chain of merged blocks.

```

1: function BLOCKLIVE( $bs$ )
2:   if  $bs.\text{merge}$  then
3:     return  $\text{BLOCKLIVE}(bs.\text{merge})$ 
4:   else
5:     return  $bs$ 

```

When the number of specializations of a basic block bb exceeds $\text{VERSION_LIMIT}(bb)$, some contexts need to be merged. Note that we express the version limit as a function of the basic block to allow the algorithm to adapt the limit to different types of basic blocks, such as

those marked by the compiler front-end as probably benefiting from more specialization. This function could simply return a constant value, as we have done in our experiments. Context merging is done by the function BLOCKMERGE SOME (Algorithm 4). It selects two versions not already merged (line 2), computes the union of the two corresponding contexts (line 3), and then replaces the merged blocks in the CFG (line 14). Merging is only triggered when a version is removed from the work queue (Algorithm 1, line 5). This allows the number of versions to temporarily exceed the version limit. This *delayed merging* increases the choices available to the selection heuristic and possibly leads to better merges.

Merging a block in the CFG entails deleting all incoming edges of the merged blocks to redirect them to the block resulting from the merge. Any deletion of an edge may render some specialized block unreachable from the CFG’s entry point. Similarly, an added edge can make some previously unreachable block reachable anew. Keeping track of unreachable blocks is required since those no longer have to be traversed and must not be considered when selecting versions to merge in BLOCKMERGE SOME (line 2). This can be done efficiently by maintaining an Even-Shiloach tree [14] of the CFG to help the SBBV algorithm detect whenever the reachability of a specialized block changes. Any block made unreachable by a merge is marked so that it is no longer considered in the merge selection and CFG traversal.

Algorithm 4 Merge blocks.

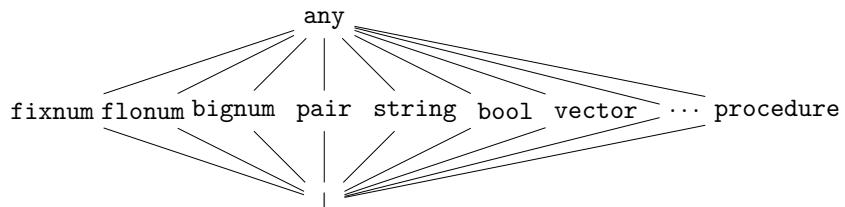
```

1: function BLOCKMERGE SOME(bb, wq)
2:   bs1, bs2  $\leftarrow \Theta^2\{\forall b \in bb.\text{versions}, \neg b.\text{merge}\}$            ▷ select two versions to merge
3:   ctx  $\leftarrow bs_1.\text{ctx} \sqcup bs_2.\text{ctx}$            ▷ merge the two corresponding contexts
4:   bs  $\leftarrow \text{BLOCKNEWVERSION}(bb, ctx, wq)$            ▷ create a new block for the merge
5:   if bs1  $\equiv$  bs then           ▷ replace the merged blocks
6:     | BLOCKMERGEANDREPLACE(bs2, bs)
7:   else if bs2  $\equiv$  bs then
8:     | BLOCKMERGEANDREPLACE(bs1, bs)
9:   else
10:    | BLOCKMERGEANDREPLACE(bs1, bs)
11:    | BLOCKMERGEANDREPLACE(bs2, bs)
12: function BLOCKMERGEANDREPLACE(obs, mbs)
13:   | obs.merge  $\leftarrow mbs$            ▷ mark that obs is merged into mbs
14:   | replace obs with mbs in the CFG           ▷ patch the CFG

```

The operator Θ^2 selects two unmerged specializations for the block *bb*. For the sake of the correctness of the algorithm, this operator might select any two versions. For instance, it could select two random versions. Of course, a better operator would positively impact the result of the compilation. In Section 4.3 we present the operator we have used so far.

The behavior of the merge operator \sqcup is independent of the SBBV algorithm but it must produce a new context that at least encompasses the contexts of the merged blocks. The natural solution is to use a lattice for organizing the information associated with variables and to go up some level for each merge. We will use the following lattice:



28:8 Static Basic Block Versioning

Note that a number can be a `flonum` (floating point numbers), a `fixnum` (small integers that fit in a machine word), or a `bignum` (integers that don't fit in a machine word). While this would be a good match for JavaScript numerical types, for a full Scheme, or Python, implementation there would also be a representation for rational and complex numbers, but we will ignore them to simplify the discussion.

Consider the merge of two contexts mapping a variable v respectively to the types `fixnum` and `flonum`. The merge operation should produce a context mapping v to `any`. In Section 3.2 we show the use of a more precise lattice for representing properties.

The last part of the SBBV algorithm is in charge of specializing blocks. It merely creates a new block where the instructions have been specialized one by one.

Algorithm 5 Specialize a block and its instructions.

```

1: procedure BLOCKSPECIALIZE( $bs, wq$ )
2:    $bb \leftarrow bs.bb$ 
3:    $ctx \leftarrow bs.ctx$ 
4:   for all  $i \in bb.instrs$  do
5:      $(ni, nctx) \leftarrow \text{INSSPECIALIZE}(i, ctx, wq)$ 
6:      $bs.instrs \leftarrow bs.instrs + ni$ 
7:    $ctx \leftarrow nctx$ 
```

Specializing an instruction that implements a type test produces a new context. For instance, when specializing the block #12 of Figure 1a, in the positive branch, the argument `x` is known to be a pair. The block #4 is then specialized with a context that reflects that information, which is then propagated to following blocks. Conversely, on the negative branch, the argument `x` is known not to be a pair. This information too, is propagated to following blocks. To handle these evolutions of the contexts, the procedure `BLOCKSPcialize` updates the context it uses for specializing the instructions after each iteration (Algorithm 5, line 7).

The function `INSSPECIALIZE` selects a specializer appropriate for the instruction.

Algorithm 6 Specialization of an instruction.

```

1: function INSSPECIALIZE( $i, ctx, wq$ )
2:   if  $i.kind$  is "goto" then return INSSPECIALIZEGOTO( $i, ctx, wq$ )
3:   else if  $i.kind$  is "if" then return INSSPECIALIZEIF( $i, ctx, wq$ )
4:   else if  $i.kind$  is ... then ...
5:   else return ( $i, ctx$ )
```

The specialization of a `goto` instruction mostly consists in forwarding the specialization context `ctx` to the target of the instruction. For instance, when specializing the `goto` in block #1 of Figure 1a with the context $\{x \mapsto \text{pair}, p \mapsto \text{procedure}\}$ the `goto` instruction triggers the specialization of the block #12 with the same context. The instruction brings no new knowledge about the variables types so it returns an unmodified context.

Algorithm 7 Specialization of `goto` instructions.

```

1: function INSSPECIALIZEGOTO( $i, ctx, wq$ )
2:    $ni \leftarrow i.dup()$ 
3:    $ni.target \leftarrow \text{BLOCKNEWVERSION}(i.target, ctx, wq)$ 
4:   return ( $ni, ctx$ )
```

The specialization of an `if` is more involved because this is where new knowledge is acquired and where requests for new specializations are emitted. If the test of the expression is not a type test, the specialization behaves as the specialization of a `goto` instruction (Algorithm 8, line 2). If the instruction implements a type check and if the context is such that the test always succeeds, then the instruction is replaced with a `goto` instruction which directly branches to the positive block (line 6). This is illustrated by the specialization of the block #6 of Figure 1a into the block #23 of Figure 1b. Conversely, if the test is known to evaluate to false, the instruction is replaced with a `nop` instruction.

Algorithm 8 Specialization of `if` instructions.

```

1: function INSSPECIALIZEIF( $i, ctx, wq$ )
2:   if  $\neg i.\text{test}$  is a typecheck then
3:      $ni \leftarrow i.\text{dup}()$ 
4:      $ni.\text{target} \leftarrow \text{BLOCKNEWVERSION}(i.\text{target}, ctx, wq)$ 
5:     return ( $ni, ctx$ )
6:   else if  $ctx.\text{types.isTrue}(i.\text{test})$  then
7:      $ni \leftarrow \text{new insGoto}(i.\text{target})$ 
8:     return INSSPECIALIZE( $ni, ctx, wq$ )
9:   else if  $ctx.\text{types.isFalse}(i.\text{test})$  then
10:     $ni \leftarrow \text{new insNop}()$ 
11:    return ( $ni, ctx$ )
12:   else
13:      $ni \leftarrow i.\text{dup}()$ 
14:      $ctx^+ \leftarrow ctx \cup \{ i.\text{test.var} \mapsto i.\text{test.type} \}$ 
15:      $ctx^- \leftarrow ctx \cup \{ i.\text{test.var} \mapsto \neg i.\text{test.type} \}$ 
16:      $ni.\text{target} \leftarrow \text{BLOCKNEWVERSION}(i.\text{target}, ctx^+, wq)$ 
17:     return ( $ni, ctx^-$ )

```

The most interesting situation is when the result of the type test cannot be inferred from the current context (line 12). In that case, two new contexts are created, in accordance to the narrowing rules for that test. The positive branch of the test will be specialized with a context reflecting the success of the type test and the context reflecting a negative result is returned to the procedure `BLOCKSPARLIZE` (for instance, see the block #12 of Figure 1a that creates the context of the block #19 of Figure 1b).

3 Improved Specializations

In Section 2 we have presented the general SBBV algorithm. We have exposed its principles that we have illustrated with a simple type analysis that maps variable usages to types. In this section, we show how the algorithm can be extended to specialize the blocks according to more fine-grained information.

3.1 Variable Aliasing

The contexts used in the example in Figure 2 enables SBBV to specialize blocks according to the type of the variables. However, it does not keep track of variable aliases, which jeopardizes the benefit of the optimization. Compilers tend to introduce many temporaries for evaluating expressions and if these aliases are not handled efficiently by the SBBV specialization, what is learned about a variable's value will not be propagated to the other

context: ctx, ctx_0, \dots

- types: a list of mappings of variable to type information
- equiv: a list of equivalence classes

Figure 3 Extended specialization contexts with variable class equivalence.

variables containing the same value. Thankfully, handling aliases merely requires extending the definition of the contexts and to handle the specialization of the `mov` instruction, which assigns a value to a variable. The new definition of the contexts is extended into that of Figure 3. The specialization of the `mov` instruction, that copies a variable into another and that is represented by the \leftarrow operator in the CFGs, is given in Algorithm 9. For the sake of simplicity, this extension keeps track of aliasing of read-only variables only. Assigned variables are never treated as aliases of other variables. Also, not presented here, the specialization of the `if` instruction (Algorithm 8) is modified so that it also propagates the gathered type information to the variable's aliases.

Algorithm 9 Specialization of `mov` instructions.

```

1: function INSSPECIALIZEMOV( $i, ctx, wq$ )
2:    $nctx \leftarrow ctx$ 
3:    $nctx.equiv[i.target] \leftarrow \emptyset$ 
4:   if  $i.source$  is a read-only variable then
5:      $nctx.equiv[i.target] \leftarrow \{i.source\}$ 
6:   return ( $i, nctx$ )

```

3.2 Specialization of Arithmetic Operations

The SBBV algorithm is general-purpose and it can be applied to other properties of the variables and values to go beyond type check removal. In this section we show how to leverage this flexibility to also specialize the basic blocks according to fixnum integer intervals `LO..HI`, where `LO` and `HI` are values in the fixnum range. This will allow the compiler to generate code using fixnum arithmetic operators that are fast (because they directly map to machine instructions) and removing overflow checks and bound checks.

The benefits of this extension can be illustrated on the expression `(+ x 1)`, which adds 1 to `x`, a very common operation in most programs. The specific operation executed depends on the type of `x`. The result could be a fixnum, a flonum, a bignum, or the operation could raise an exception if `x` is not a numerical type. Moreover, the result of `(+ x 1)` will be a bignum if `x` is maxfix, the largest fixnum value. A similar dispatch is part of the semantics of most arithmetic operators `(-, *, ...)` and comparison operators `(=, <, ...)`, and in the general case, such as `(+ x y)`, the dispatch is on the combination of types of `x` and `y`.

Optimizing compilers usually inline the handling of the most common cases, such as all operands being fixnums, and all operands being flonums, and defer the handling of the remaining cases to an out-of-line function. For example, the expression `(+ x 1)` could be expanded by the compiler to this code:

```

(if ($fixnum? x)
  (or ($fix+? x 1)    ;; fixnum add 1 with overflow check (#f returned on overflow)
      ($+ x 1))       ;; call $+ function to handle bignum result case
  (if ($flonum? x)
      ($fl+ x 1.0)    ;; flonum add 1
      ($+ x 1)))      ;; call $+ function to handle other cases including errors

```

Here we use names prefixed with ‘\$’ to indicate internal operations of the system:

- $(\$fixnum? \ x)$ and $(\$flonum? \ x)$ test to see if x is a fixnum, or a flonum respectively.
- $(\$+ \ x \ y)$ is an addition function in the runtime system that handles all possible type combinations for x and y , including those that raise an exception.
- $(\$fl+ \ x \ y)$ adds two flonums to give a flonum result.
- $(\$fx+? \ x \ y)$ adds two fixnums to give a fixnum result or false in the case of an overflow. This operation includes an overflow check that makes it somewhat more expensive than $(\$fx+ \ x \ y)$ that does not check for overflow (note the absence of the trailing ‘?’).

When the CFG of the above expansion of $(+ \ x \ 1)$ is processed by the SBBV algorithm various optimizations can happen. If the context indicates that x is a fixnum then the $(\$fixnum? \ x)$ test in the specialized basic block is an unconditional jump to the CFG of $(\text{or} \ (\$fx+? \ x \ 1) \ (\$+ \ x \ 1))$, effectively removing the code that handles flonums, bignums, and other types. Moreover, if it is known that x is in the interval LO..HI where $\text{HI} < \text{maxfix}$, then the result of $(\$fx+? \ x \ 1)$ is in the fixnum interval $\text{LO} + 1..\text{HI} + 1$, so an overflow is impossible, i.e., $(\$fx+? \ x \ 1)$ is necessarily a fixnum. Consequently the CFG can be specialized to $(\$fx+ \ x \ 1)$, which is a machine integer addition with no overflow check.

Other languages, such as Ruby, support similar generic arithmetic, but more importantly, languages such as JavaScript [16] and Python that do not expose small integers require that the compiler be able to detect when operations can be implemented as fixnum operations. Hence, these fast implementations must be able to detect when an operation overflows and, exactly as Scheme does, promote the number in such a case (JavaScript promotes them to IEEE floating point numbers, Python to bignums).

To extend the analysis, we refine the definition of specialization contexts (Figure 4) and we refine the lattice of values (Figure 5). Fixnum values are now represented with intervals.

Handling numerical values requires us to modify the specialization of the *if* and the *fixnum arithmetic with overflow* instructions, such as $\$fx+?$. The first one must compute new interval approximations to be propagated in the positive and negative branches by using interval narrowing techniques [11]. The second one must implement arithmetic operations over intervals [23], such as $x_{\text{lo}}..x_{\text{hi}} + y_{\text{lo}}..y_{\text{hi}} = (x_{\text{lo}} + y_{\text{lo}})..(x_{\text{hi}} + y_{\text{hi}})$.

Algorithm 10 Specialization of *if* instructions with numerical values.

```

1: function INSSPECIALIZEIF( $i, ctx, wq$ )
2:   if  $i$  is an integer comparison then
3:      $(ctx^+, ctx^-) \leftarrow \text{intervalNarrowing}(i, ctx)$ 
4:      $ni \leftarrow i.\text{dup}()$ 
5:      $ni.\text{target} \leftarrow \text{BLOCKNEWVERSION}(i.\text{target}, ctx^+, wq)$ 
6:     return  $(ni, ctx^-)$ 
7:   else
8:     as in algorithm 8

```

For instance, let us assume the specialization of the instruction “*if* $(\$fx> \ i \ 3)$ ” in a context $\{i \mapsto \text{minfix}..\text{10}\}$. The new INSSPECIALIZEIF will generate the two new contexts $\{i \mapsto 4..\text{10}\}$ and $\{i \mapsto \text{minfix}..3\}$ for the positive and negative outcomes respectively.

The numerical operation specialization replaces a numerical operator, such as $\$fx+?$, whose result is known not to overflow with a faster operator that does not check for overflow, such as $\$fx+$. It must also implement some widening operation [10] in order to hasten the convergence of the algorithm. Without widening the algorithm would require a number of iterations proportional to the size of the interval representing numbers, which would be prohibitive. The widening is handled by the \sqcup operator of the Algorithm 4 (see Section 2.2).

28:12 Static Basic Block Versioning

context: ctx, ctx_0, \dots

- types: a list of mappings of variable to type information
- equiv: a list of equivalence classes
- range: a list of mappings of variable to integer intervals

Figure 4 Specialization contexts extended with integer intervals.

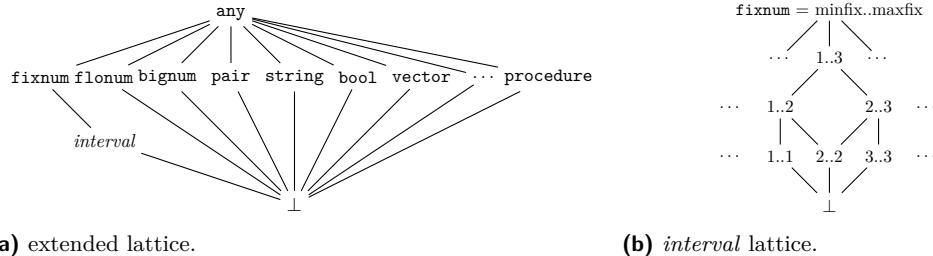


Figure 5 Extending the lattice used for variable values to handle integer intervals.

3.3 Vector Support

The previous section showed how to extend SBBV to approximate integer values as intervals described with two integer bounds. While this enables the compiler to remove overflow checks, it is insufficient to remove bound checks of vector accesses as the length of a vector is generally unknown statically and cannot be represented with an exact integer value.

To handle vectors we introduce a refined representation of intervals. With this extension interval bounds can be represented with integers as before, but also with the symbolic value $\llbracket v \rrbracket - i$ that is equal to the length of vector v minus the integer offset $i \geq 0$. Here we assume that the length of a vector is always a nonnegative fixnum. For instance, the interval $0..\llbracket v \rrbracket - 1$ denotes all the nonnegative fixnums that are lower than the length of the vector v , in other words, the valid indexes of vector v . The upper bound of that interval, *i.e.*, $\llbracket v \rrbracket - 1$, itself denotes a fixnum in the interval $-1..maxfix - 1$.

We illustrate the benefit of this extension to SBBV with a variant of the `find` function presented in Section 2 that operates on vectors and lists:

```

1  (define (findv p x)
2    (if (vector? x)
3        (let ((len (vector-length x)))
4            (let loop ((i 0))
5                (if (< i len)
6                    (let ((e (vector-ref x i)))
7                        (if (p e)
8                            e
9                            (loop (+ i 1)))))
10                   #f)))
11    (find p x)))

```

```

1 (define (findv p x)
2   (if ($vector? x)
3     (let ((len (if ($vector? x) ($vector-length x) ($fail))))
4       (let loop ((i 0))
5         (if (and ($fixnum? i) ($fixnum? len))
6             ($fx< i len)
7             (if (and ($flonum? i) ($flonum? len))
8                 ($fl< i len)
9                 ($< i len)))
10            (let ((e (if (and ($vector? x) ($fixnum? i))
11                  ($fx>= i 0) ($fx< i ($vector-length x)))
12                  ($vector-ref x i)
13                  ($fail)))
14              (if ((if ($procedure? p) p ($fail)) e)
15                  e
16                  (loop (if (and ($fixnum? i) ($fixnum? 1))
17                          (or ($fx+? i 1) ($+ i 1))
18                          (if (and ($flonum? i) ($flonum? 1))
19                              ($fl+ i 1)
20                              ($+ i 1)))))))
21            #f)))
22      (find p x)))

```

Figure 6 The code of the `findv` function where all dynamic checks are explicit.

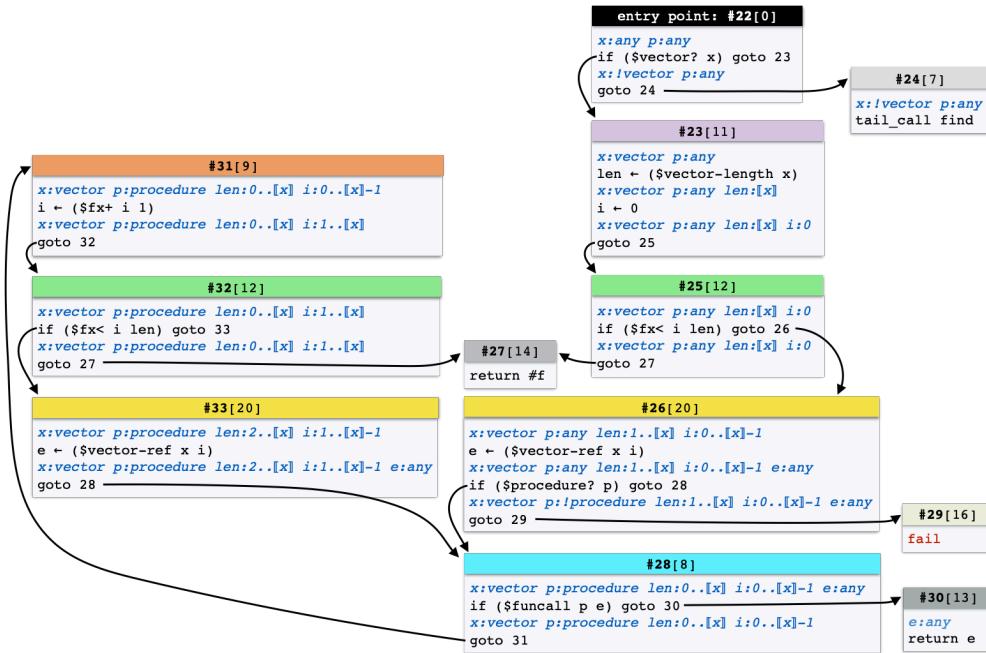


Figure 7 The specialized CFG of the `findv` function showing that the index calculations are done entirely with fixnums with no overflow checks and no vector bound checks. For brevity, we write $\llbracket x \rrbracket$ instead of $\llbracket x \rrbracket - 0$, and the interval $\llbracket x \rrbracket .. \llbracket x \rrbracket$ is abbreviated to $\llbracket x \rrbracket$.

Figure 6 shows the code after the compiler has blindly expanded each operation to include all required dynamic checks. The refinements of SBBV presented in this section enables the compiler to create the specialized CFG shown in Figure 7, which is optimal (all bound checks and overflow checks have been removed). There is only a procedure check for parameter p in the first iteration of the loop, and only if parameter x is a non-empty vector.

This refinement does not impact the SBBV algorithm nor does it demand to change the specialization of the arithmetic instructions but it requires us to extend the interval operators to treat cases where at least one bound is a symbolic value. For example, in the case of the interval addition $x_{\text{lo}}..x_{\text{hi}} + y_{\text{lo}}..y_{\text{hi}} = (x_{\text{lo}} + y_{\text{lo}})..(x_{\text{hi}} + y_{\text{hi}})$, the addition of the lower bounds ($x_{\text{lo}} + y_{\text{lo}}$) is computed from the following rules, where i and j denote integer values, and v and w denote vector identifiers:

$$\begin{aligned} i +_{\text{lo}} j &\mapsto i + j \\ ([v] - i) +_{\text{lo}} j &\mapsto j - i \\ j +_{\text{lo}} ([v] - i) &\mapsto ([v] - i) +_{\text{lo}} j \\ ([v] - i) +_{\text{lo}} ([w] - j) &\mapsto -i - j \end{aligned}$$

and the addition of the upper bounds ($x_{\text{hi}} + y_{\text{hi}}$) is computed from the following rules, where *overflow* denotes an upper bound that is not a fixnum:

$$\begin{aligned} i +_{\text{hi}} j &\mapsto i + j \\ ([v] - i) +_{\text{hi}} j &\mapsto [v] - (i - j) \quad \text{if } i \geq j \\ ([v] - i) +_{\text{hi}} j &\mapsto \text{overflow} \quad \text{if } i < j \\ j +_{\text{hi}} ([v] - i) &\mapsto ([v] - i) +_{\text{hi}} j \\ ([v] - i) +_{\text{hi}} ([w] - j) &\mapsto \text{overflow} \end{aligned}$$

The narrowing operations for comparisons are similar to that of regular intervals, considering that vector lengths are themselves modelled as intervals from 0..maxfix. Below are the rules for the narrowing of $x < y$ from which the rules for the other comparisons can be derived:

Narrowing rule for: $x < y$ with $\{x \mapsto x_{\text{lo}}..x_{\text{hi}}, y \mapsto y_{\text{lo}}..y_{\text{hi}}\}$

Positive outcome: $\{x \mapsto x_{\text{lo}}.. \min_{\text{hi}}(x_{\text{hi}}, y_{\text{hi}} - 1), y \mapsto \max_{\text{lo}}(x_{\text{lo}} + 1, y_{\text{lo}})..y_{\text{hi}}\}$

Negative outcome: $\{x \mapsto \max_{\text{lo}}(x_{\text{lo}}, y_{\text{lo}})..x_{\text{hi}}, y \mapsto y_{\text{lo}}.. \min_{\text{hi}}(x_{\text{hi}}, y_{\text{hi}})\}$

$$\begin{aligned} \max_{\text{lo}}(x, y) &\mapsto x \text{ if } \text{val}_{\text{lo}}(x) > \text{val}_{\text{lo}}(y) \text{ else } y \\ \min_{\text{hi}}(x, y) &\mapsto x \text{ if } \text{val}_{\text{hi}}(x) < \text{val}_{\text{hi}}(y) \text{ else } y \end{aligned}$$

$$\begin{aligned} \text{val}_{\text{lo}}(i) &\mapsto i \\ \text{val}_{\text{lo}}([v] - i) &\mapsto i \end{aligned}$$

$$\begin{aligned} \text{val}_{\text{hi}}(i) &\mapsto i \\ \text{val}_{\text{hi}}([v] - i) &\mapsto \text{maxfix} - i \end{aligned}$$

It is noteworthy that, as a result of the interval widening, some of the inferred intervals are somewhat conservative (for example at block #28 the interval for `len` could have been $1..[v]$). The widening loses some information but makes computing a fix-point faster.

4 Experiments

In this section we demonstrate the practicality of SBBV through experiments. To ensure that our results are not overly system specific, we have integrated an SBBV pass in the compilation pipeline of two existing Scheme compilers, Bigloo [19] and Gambit [20]. Both

of these are independently developed mature optimizing AOT Scheme to C compilers that use a CFG representation of the compiled program. Moreover these compilers are used as back-ends of optimizing compilers for JavaScript [31, 30] and Python [22]. Bigloo and Gambit implement a slew of features and classical optimizations such as constant-folding, function inlining, flat closures, and lambda-lifting. Any performance improvements would constitute a notable achievement given the many years of fine-tuning that went into their development. We put this in perspective in Section 4.5.

Adding SBBV to these compilers allows them to use the type, range and value of variables to perform flow sensitive code specialization that is tailored to the program logic, with low code bloat. The experiments have been designed to demonstrate further performance improvements by SBBV than by other optimization techniques implemented by these compilers.

We evaluate the impact of SBBV by applying it to a suite of benchmarks. Each benchmark is compiled with and without SBBV to measure its impact on the number of dynamic checks, program size, execution time, and compilation time. Section 4.1 provides a brief description of the benchmark suite. Benchmarks were executed on a machine with an Intel Core i7-7700K, 48 GB of RAM, and under Debian 10.13 with kernel version SMP Debian 4.19.269-1.

In order to measure SBBV’s impact on the number of dynamic checks, both compilers have been instrumented to count the number of dynamic checks during a program execution. Executions for measuring time and dynamic checks are done separately to ensure that counting checks does not affect the measured execution time. Execution time is measured by profiling each executable with “`perf stat`” to measure its execution real-time. Each benchmark is parameterized such that its execution lasts at least five seconds on our machine, and is repeated 50 times, removing the top and bottom 5 outliers. The parameters of each benchmark are provided as command-line arguments to ensure that the compiler does not optimize for specific values or types. All timing results in this section are the average execution time of each benchmark. The relative standard deviation of the execution time never exceeds 0.24% on macrobenchmarks, and 2.20% on microbenchmarks; consequently we omit standard deviations in figures to improve readability.

4.1 Benchmark Programs

Our benchmark suite combines programs from two sources: the R7RS benchmark suite [1] that is commonly used for evaluating the performance of Scheme systems, and benchmark programs used in [30] that have Scheme and JavaScript versions.

We use both macrobenchmarks and microbenchmarks, which we classify according to their size (fewer than 150 lines of code is a microbenchmark). These two classes are distinguished because microbenchmarks stress a narrow set of features and consequently are poor predictors of the overall performance of a system. We only use microbenchmarks as instruments for shedding light on specific behaviors of the SBBV algorithm.

Here is a brief description of the benchmark programs:

Macrobenchmarks:

- `almabench` (430 LOC): Compute the celestial coordinates of the sun at noon. Uses floating point numbers, vectors, and assignments.
- `boyer` (610 LOC): Prolog-like rule-directed rewriting engine. Uses pairs and symbols.
- `compiler` (11,740 LOC): Old version of the Gambit Scheme compiler generating M68000 code. Uses pairs, symbols, vectors, and strings.
- `conform` (490 LOC): Graph type checker using equivalence classes. Uses lists and strings.

28:16 Static Basic Block Versioning

- **dynamic** (2,350 LOC): Dynamic type inference for Scheme. Uses lists, symbols, and higher-order functions.
- **earley** (660 LOC): Earley parser parsing an ambiguous grammar. Uses vectors, lists, small integers and symbols.
- **l eval** (560 LOC): Scheme interpreter based on closures. Uses lists, symbols, and higher-order functions.
- **maze** (740 LOC): Hexagonal grid maze generator. Uses vectors and small integers.
- **nucleic** (3,510 LOC): 3D structure determination of a nucleic acid. Uses vectors and floating point numbers.
- **peval** (630 LOC): Partial evaluator for Scheme. Uses lists, symbols, and higher-order functions.
- **scheme** (1,090 LOC): Other Scheme interpreter based on closures. Uses lists, symbols, and higher-order functions.
- **slatex** (2,470 LOC): Scheme to Latex processor. Uses characters, strings, lists, vectors, and small integers.

Microbenchmarks:

- **ack** (10 LOC): Ackermann function. Uses small integers and recursion.
- **bague** (110 LOC): Solver of the *baguenaudier* puzzle. Uses small integers and vectors.
- **fib** (20 LOC): Fibonacci function. Uses small integers and recursion.
- **fibfp** (20 LOC): Fibonacci function. Uses floating point numbers and recursion.
- **nqueens** (40 LOC): Solver of the N-queens puzzle. Uses lists, small integers, and recursion.
- **primes** (40 LOC): Sieve algorithm for finding primes. Uses lists and small integers.
- **tak** (20 LOC): Takeuchi function. Uses small integers and recursion.

4.2 Counting Dynamic Checks

In the Bigloo and Gambit implementations, many built-in procedures implicitly check the type of their arguments and signal an error if they are invalid, such as arithmetic on non-number types or index out of bound when indexing a vector. Polymorphic operators also use implicit dynamic type checks to dispatch computation to specialized primitives.

To measure dynamic checks, all built-in procedures used in our benchmarks are redefined with macros that use inline checks. This is semantically equivalent to operations that apply checks and dispatch to specialized primitives implicitly. For instance, the `BBVvector-ref` and `BBV+` macros implement the `vector-ref` and `+` operations respectively:

```
1 (define-macro (BBVvector-ref v i)
2   '(let ((v ,v) (i ,i))
3     (if (and ($vector? v) ($fixnum? i)
4              ($fx>= i 0) ($fx< i ($vector-length v)))
5         ($vector-ref v i)
6         (error "vector-ref error")))))
7
8 (define-macro (BBV+ x y)
9   '(let ((x ,x) (y ,y))
10    (if (and ($fixnum? x) ($fixnum? y))
11        (or ($fx+? x y) ($+ x y))
12        (if (and ($flonum? x) ($flonum? y))
13            ($f1+ x y)
14            ($+ x y)))))
```

The macros use specialized operators (prefixed with \$ in the example), making all checks explicit (type checks, array bound checks, and integer overflow checks) and ensuring that both compilers perform the same set of checks and in the same order (see Section 3.2 for the definitions of `$fx+?`, `$+`, and the other primitive operations).

When SBBV has determined the type of a value, the type tests that are in the expansion of these macros (`$fixnum?`, `$flonum?`, etc) are effectively removed. Similarly, SBBV’s interval analysis may determine the range of possible integer values, allowing comparisons such as `($fx>= i 0)` to be removed, and calls to overflow checking operators such as `($fx+? x y)` to be replaced by the non-overflow checking `($fx+ x y)` when the result cannot overflow.

However, not all dynamic checks originate from safe operators since programmers can add type tests and bound checks as part of their program’s logic. For this reason, we distinguish between checks introduced by safe operators, which we call *safety* checks in the context of this experiment, and those introduced by the programmer. The number of safety checks is computed by replacing all operators, such as `BBVvector-ref` and `BBV+`, by unsafe ones that execute no type tests, overflow checks or array bound checks. By subtracting the number of checks executed by the unsafe version of a program from the number of checks of its safe version, we obtain the number of safety checks.

4.3 Merge Selection

Applying SBBV requires a heuristic for selecting which versions of a block to merge when that block’s version limit is exceeded. This corresponds to choosing a concrete implementation for the Θ^2 operator from Algorithm 4. The quality of the selection function impacts the general performance of the SBBV algorithm. The current Bigloo and Gambit implementations use a merge heuristic that is rudimentary but still sufficient to establish the benefit of the approach.

In our implementations, when the version limit of a basic block is exceeded, versions that are the most similar are merged first. The similarity of two versions is computed by counting how many variables have the same type when entering the block. While both Bigloo and Gambit implement a *selection by similarity*, the exact implementations of their selection functions differ slightly due to how they represent types internally.

Given that the versions selected depend solely on the contexts when entering a block, this is a *local* merge heuristic. It requires no usage analysis of each variable in the block and its successors. A nonlocal merge heuristic could lead to better results if it prioritizes some versions over others with the objective of maximizing the benefits for the whole program. The space of possible merge heuristics is large and we intend to explore it in future work.

4.4 Results

We first present broad results before diving into a deeper analysis in the subsequent sections.

SBBV permits a trade-off between code size and dynamic checks removal. As the version limit increases, more blocks are duplicated and more dynamic checks are removed. This comes at the cost of increased executable size and compilation time. We found a limit of 2 versions to offer a good trade-off between checks removal, size, and compilation time.

Figure 8 shows the proportion of safety checks removed by SBBV, such as type, overflow, and array bound checks. In all benchmarks, the number of checks decreases when compared to the executable without SBBV. This indicates that SBBV can remove dynamic checks that the existing optimizations of Bigloo and Gambit could not remove.

The proportion of checks removed varies between Bigloo and Gambit, despite both compilers applying the same SBBV algorithm. Bigloo removes more checks without applying SBBV, thus leaving fewer checks to be removed by SBBV. However, the absolute number of remaining checks is similar between both implementations. To a lesser extent, differences in the implementations of the heuristic for selecting versions to merge also influence the number of removed checks by each compiler.

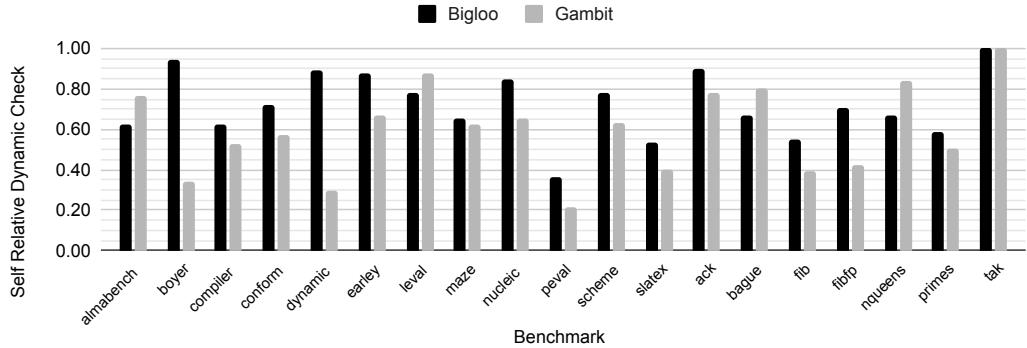


Figure 8 Relative number of safety checks executed for benchmarks compiled with and without SBBV (limit of 2 versions), separately for Bigloo and Gambit. 1.0 corresponds to compilation without SBBV. Lower values indicate fewer dynamic checks with SBBV.

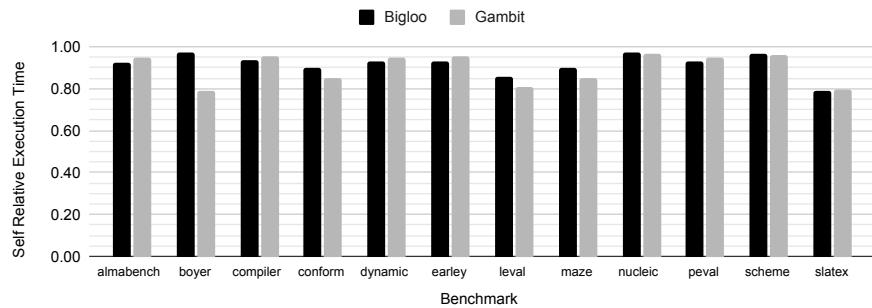


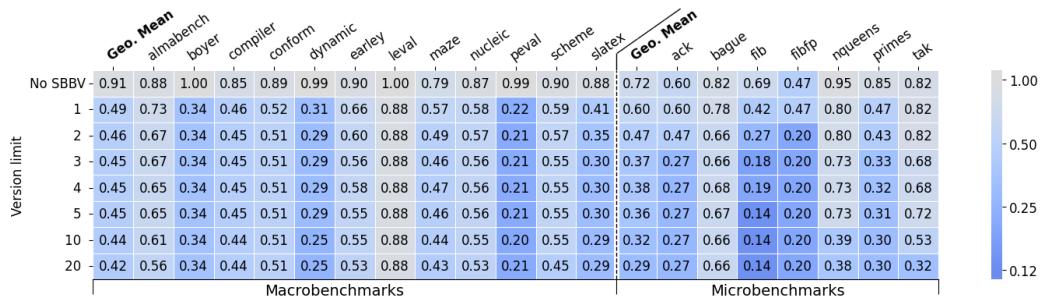
Figure 9 Relative execution time of macrobenchmarks compiled with and without SBBV (limit of 2 versions), separately for Bigloo and Gambit. 1.0 corresponds to compilation without SBBV. Lower values indicate better performance with SBBV.

Although the number of dynamic checks decreases with higher version limits, it does not always lead to a similar reduction of the execution time. Figure 9 shows that all macrobenchmarks execute faster with SBBV and a limit of 2 version (by 10% on average). However, no significant speedup is observed by further increasing the version limit. The relation between the version limit and execution time is discussed further in Section 4.4.3.

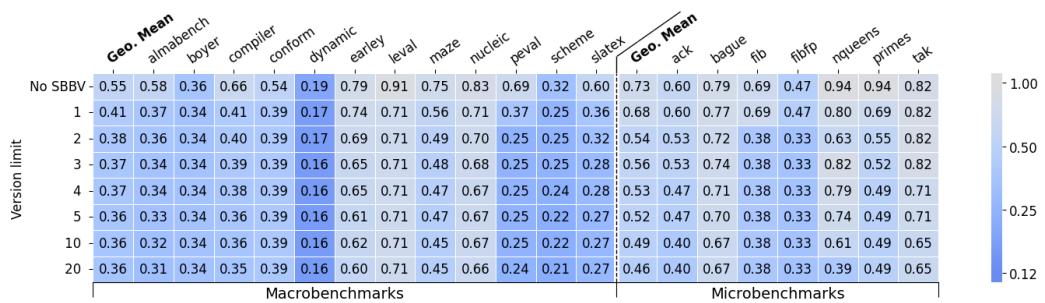
Increasing the version limit allows for more specialized versions. In the following sections, we take into account the impact of the version limit on the removal of dynamic checks, program size, execution speed, and compilation time. Each benchmark is compiled with version limits ranging from 1 to 5, as well as with limits of 10 and 20 versions, and it is compared to a compilation without SBBV. Limits higher than 5 are probably not very practical due to diminishing returns for the added compilation time. We tested with limits of 10 and 20 versions mostly to check the performance in extreme cases.

4.4.1 Dynamic Checks

Figure 10 shows the proportion of safety checks remaining after SBBV with increasing version limits. We estimate the number of remaining safety checks by subtracting checks in the unsafe version of a benchmark from those in the benchmark compiled with SBBV. To compute the total number of safety checks without SBBV, we apply the same formula to each benchmark compiled with no optimization, which effectively preserves all checks.



(a) Gambit.



(b) Bigloo.

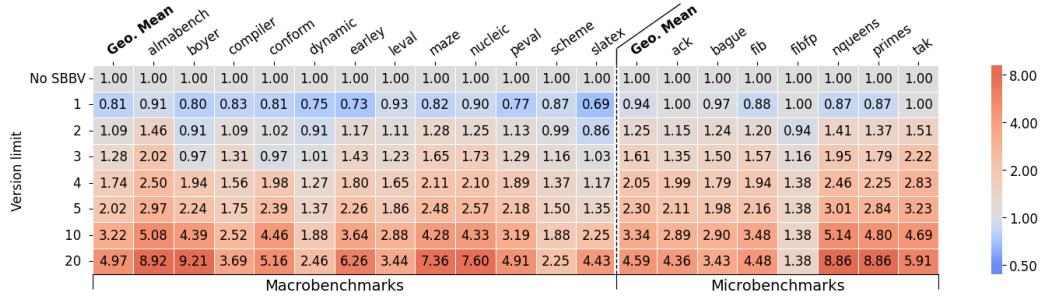
Figure 10 Effect of SBBV on the removal of dynamic checks (type, overflow, and array bound checks) with increasing version limits. Each cell shows the proportion of safety checks remaining for a given version limit and benchmark when compared to an unoptimized execution. The first row shows checks when SBBV is not applied and only existing optimization techniques are used. A ratio of 1 means that no dynamic checks were removed. Lower values indicate that more checks were removed.

Figures 10a and 10b show, on the first row, the proportion of remaining checks after applying the standard Gambit and Bigloo optimization techniques (*No SBBV*). The following rows display results with SBBV and specific version limits. For all benchmarks, SBBV removes more checks than the standard optimizations. As the version limit increases, more specialized versions are generated, allowing removal of additional checks.

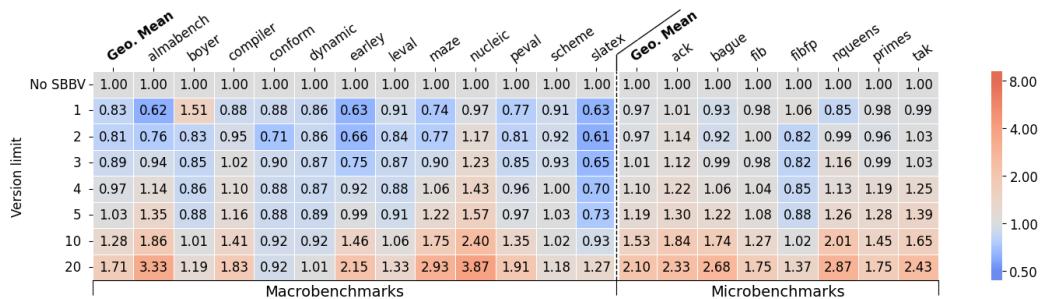
For low version limits, the number of dynamic checks steeply decreases as the limit increases. However, beyond a version limit of about 4, there is a diminishing return for almost all benchmarks. In some benchmarks, an upper bound is rapidly reached beyond which almost no more checks are removed (such as `boyer` at 1 version). In these cases, further increasing the limit contributes to the generation of relatively unimportant versions. Conversely, new useful versions are still discovered when increasing the version limit beyond 10 for some benchmarks (such as `tak` with Gambit).

Increasing the version limit sometimes increases the number of dynamic checks. For instance, in Figure 10a, the number of dynamic checks increases when incrementing the version limit from 2 to 3 in the `nqueens` benchmark with Bigloo. Increasing the version limit delays the merge of excess versions until more candidates are discovered. Given additional choices, the selection function may choose differently, merging a useful version that would have been kept otherwise. This highlights the room for improvement of our selection function.

28:20 Static Basic Block Versioning



(a) Gambit.



(b) Bigloo.

Figure 11 Program size with increasing version limits, relative to using the standard optimizations (No SBBV). Each cell shows the ratio between the program size with and without SBBV for a given version limit and benchmark. Lower values are better.

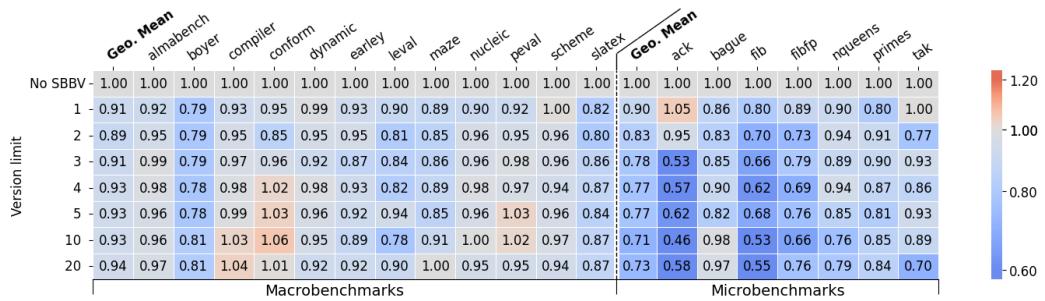
4.4.2 Program Size

We measured the program size of benchmarks compiled with SBBV. The size in bytes of each benchmark is obtained by disassembling its executable and subtracting the position of compiler specific labels. Hence, only the size of the code corresponding to each benchmark, excluding any runtime procedures, is considered.

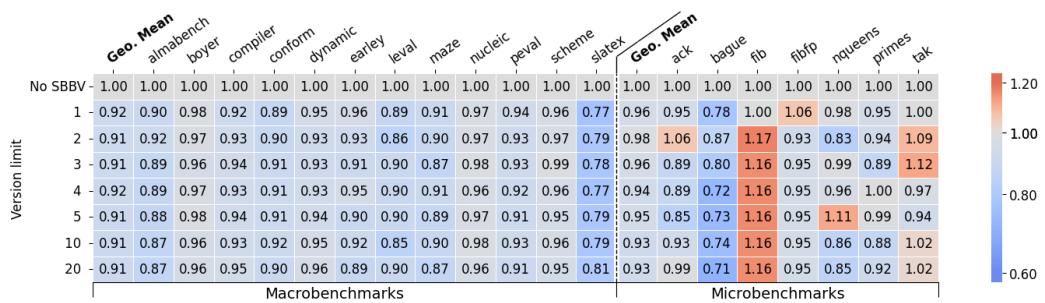
Since SBBV applies code duplication, higher version limits generally generate larger programs. Yet, low version limits may result in smaller executables. In the case of a limit of a single version, this is to be expected because SBBV becomes akin to a static type inference analysis without code duplication, which removes some unnecessary checks. However, for low enough version limits higher than one, SBBV can still reduce program size. In these cases, the removal of dynamic checks outweighs the duplication of basic blocks.

Figure 11b shows the relation between the size of a benchmark and the allotted version limit in Bigloo. On average, macrobenchmarks are smaller up to a limit of 5 versions. In general, the size increases with the version limit, but remains reasonably low with an average growth of about 1.7× on macrobenchmarks with a limit as high as 20 versions.

Figure 11a shows a similar pattern in Gambit, but with a higher growth rate. In the worst case, a growth of about 10× is observed (boyer, limit of 20 versions), highlighting the need to select a low enough version limit to curb code bloat. We found a version limit ranging from 2 to 4 to be a good compromise between removed dynamic checks and size.



(a) Gambit.



(b) Bigloo.

Figure 12 Execution time with SBBV and increasing version limits, relative to the standard optimizations (*No SBBV*). Each cell shows the ratio between the execution time with and without SBBV for a given version limit and benchmark. Lower values are better.

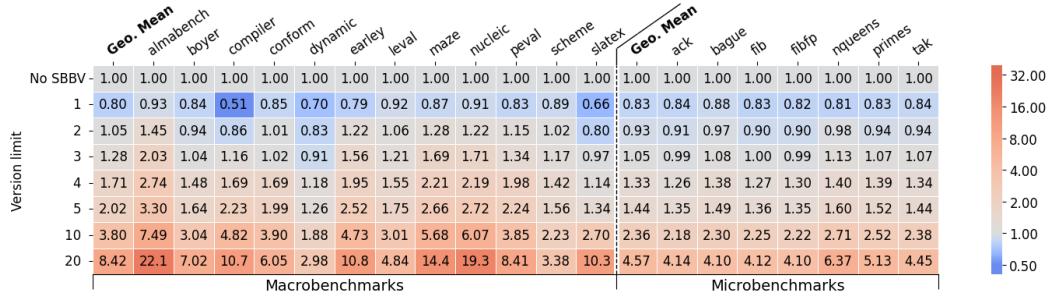
4.4.3 Execution Time

Figure 12 shows how the execution time of each benchmark varies with the version limit. Comparing performance to the number of dynamic checks from Figure 10 shows that a lower number of checks is not correlated to a faster execution in general. With Bigloo (Figure 12b), macrobenchmarks compiled with SBBV execute, on average, about 10% faster than without SBBV regardless of the version limit. With Gambit (Figure 12a), a similar speedup is observed for version limits below 4. Beyond this limit, execution speed still benefits, albeit to a lesser extent. We suspect that this discrepancy is caused by the increased code bloat observed with Gambit for high version limits, which reduces the performance of the instruction cache.

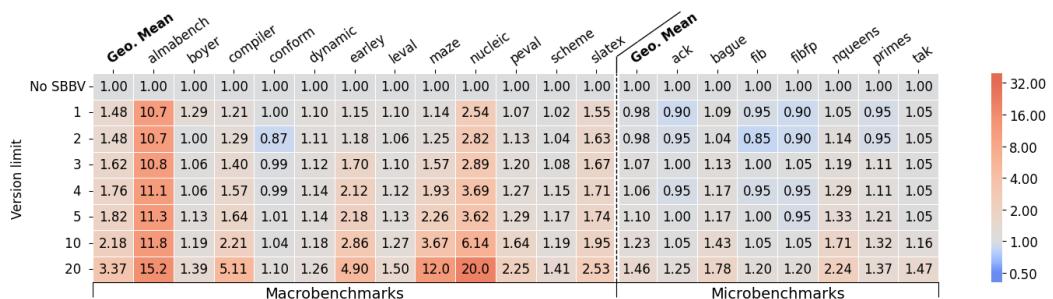
While some benchmarks benefit from a high version limit, the code speed and version limit is only vaguely correlated and contains noise. When optimizing for code speed, using a default version limit for all programs is suboptimal and it is good to give the programmer a manual control over the limit to explore the tradeoffs.

We explain this in part by hard-to-predict hardware optimizations by modern processor architectures. In particular, branch prediction makes dynamic type checking extremely cheap in typical code where a type check frequently returns the same result. Moreover, Bigloo and Gambit implement inexpensive type checks using pointer tagging. We hypothesize that SBBV would have a higher performance impact in implementations with more costly dynamic checks, such as NaN tagging, or object representations that need a memory access to check the type, such as BiBOP and object-oriented languages such as Java, Python, and Ruby.

28:22 Static Basic Block Versioning



(a) Gambit.



(b) Bigloo.

Figure 13 Compilation time with increasing version limits, relative to using the standard optimizations (No SBBV). Each cell shows the ratio between the compilation time with and without SBBV for a given version limit and benchmark. Lower values are better.

This highlights the need to refine the merge selection function. In the future, we intend to explore the space of possible merge heuristics, including nonlocal heuristics. We also wish to explore dynamic version limits, for instance by increasing the version limit of basic blocks that are likely to be in megamorphic code, as is done by YJIT [7, 8].

4.4.4 Compilation Time

We measured the compilation time for each benchmark and version limit and compared it to the compilation time without SBBV. Figure 13 shows the effect of the version limit on compilation time. In general, compilation time increases with the version limit. The reasons for this increase are twofold: firstly a higher version limit leads to more specialized versions of basic blocks within a control flow graph, secondly the increased size of the C code generated by Bigloo and Gambit leads to increased compilation time by the C compiler. Consequently, a lower program size is correlated to a shorter compilation time.

In the extreme case of a limit of 20 versions, compilation time increased by about 22× in the worst case (almbench with Gambit). However, in Section 4.4.1 we showed that there is a diminishing return from increasing the limit beyond about 4 versions. Choosing a limit of 2 caps the worst observed compilation time increase to about 1.5× with Gambit (almbench) while reaping most of the benefit of SBBV. On the same benchmark, Bigloo has a large compilation time increase starting at a limit of 1. This is due to the combined effect of a large function with multiple dispatch points and a live variable recalculation by the compiler that is not done by Gambit (register allocation is done before SBBV in the case of Gambit, and after SBBV in the case of Bigloo).

Program	Node.js 21.7.1	Bigloo 4.6a		Gambit 4.9.4-377		Chez 9.5.1	Racket 7.2
		No SBBV	SBBV	No SBBV	SBBV		
almabench	6.31	15.68	14.46	14.64	13.86	17.69	19.30
boyer	40.31	7.40	7.22	8.21	6.49	8.09	12.38
earley	56.75	14.90	13.83	10.87	10.34	9.53	24.73
leval	18.42	7.53	6.47	12.03	9.74	7.16	16.96
maze	10.07	7.47	6.70	6.75	5.73	12.01	12.12
bague	305.04	12.90	11.19	15.54	12.93	21.01	19.33

Figure 14 Average execution times of the benchmarks from [30] in seconds. Bold numbers indicate the fastest execution time for each benchmark. Lower is better.

4.5 Putting the Results in Context

The significance of the above results can be best appreciated through a comparison with other systems whose performance is more widely known. Our goal is to show that both Gambit and Bigloo are competitive with some of the leading implementations of dynamically typed languages, and thus that using SBBV is attractive in the context of high-performance implementations to increase performance further.

The Node.js system is a good comparable given that the JavaScript V8 JIT compiler on which it is based has been extensively engineered and it executes a dynamically typed programming language with similar core constructs as Scheme, to which prototype object-oriented features are added. The Chez Scheme [26] and Racket [25] systems are also interesting as high-performance representatives of the Lisp/Scheme family.

In order to do a fair comparison of systems for different languages, we use benchmark programs that have been translated to both Scheme and JavaScript in previous work [30]. Those programs are a subset of those used in the previous sections. Because of the similarity of JavaScript and Scheme, a systematic translation of the constructs is possible while avoiding important stylistic changes that would compromise the validity of the comparison. We have used the latest versions of those systems available at the time of writing through the Debian package manager (Node.js 21.7.1, Chez Scheme 9.5.1, and Racket 7.2). We measure the program execution time and, for Gambit and Bigloo, we compile the program without SBBV and with SBBV and a limit of 2 versions. Figure 14 gives the execution times in seconds.

The execution time for Gambit and Bigloo without SBBV is faster than Node.js on all the benchmarks except `almabench`, up to 24× faster for `bague`. Node.js does better on `almabench` than any other system because V8 has special optimizations for floating point numbers and arrays that are used by `almabench`. Gambit and Bigloo without SBBV are in the same ballpark as Chez Scheme, faster on roughly half the benchmarks. Racket is typically slower than Chez Scheme, on which it is built internally. When SBBV is used, both Bigloo and Gambit are consistently faster than without SBBV, including on benchmarks on which they already outperformed other compilers. This reinforces our belief that SBBV allows some optimizing compilers for dynamically typed programming language to generate even better code.

5 Related Work

Basic Block Versioning (BBV) was introduced by Chevalier-Boisvert and Feeley [5] as a technique for type check removal in JavaScript. The *lazy* BBV variant, suitable for JIT compilers, only generates versions that are executed, so it limits code bloat. On the other

hand, when the version limit is reached a fully generic version must be used, which negatively impacts type check reduction. SBBV avoids falling back on a generic version by selecting a set of versions that cover all cases but that are at least somewhat specialized. As shown in their Figure 7, a version limit of 2, which is their setting $\text{maxvers}=1$, shows a modest reduction of type checks for many benchmarks when compared to $\text{maxvers}=5$ because the fallback on the fully generic version is reached too quickly. SBBV achieves good performance with lower version limits. A variant of lazy BBV is used in production in the YJIT compiler inside CRuby [7, 8], also falling back on a fully generic version upon reaching the version limit (which can be 4, 10, or 20 depending on the situation).

The *eager* BBV variant described in [5] is suitable for AOT compilers, like SBBV, but suffers from large code bloat so it was deemed impractical and not explored further [5]. In comparison, SBBV with a version limit of 2, which achieves a comparable level of dynamic check removal, causes an average code size increase of 9% for Gambit and a decrease of 19% for Bigloo.

SBBV can be explained by the theory of abstract interpretation introduced by Cousot and Cousot [10]. The authors presented a theoretical framework for building lattice-based fixed point algorithms. Their work ensures that the SBBV algorithm converges. Furthermore, their seminal paper introduced *union with widening*. Widening not only ensures that the algorithm converges, but also that it converges fast enough to be practical.

SBBV generalizes well-known optimization techniques such as loop-unrolling [2], constant-folding [2], and tail duplication [24]. Tail duplication replicates the code after conditional branches instead of merging the control flow to a single basic block. This permits propagating the information acquired from a condition beyond the body of each branch, allowing further optimizations.

Determining when to apply tail duplication remains a challenge. Leopoldseder *et al.* proposed a simulation-based approach to determine which duplications are the most promising in term of optimization opportunities while minimizing code size [18]. This is analogous to how the choice of a selection function impacts the efficiency of SBBV.

More recently, D’Souza *et al.* applied tail duplication in TASTyTruffle, a Scala JIT compiler using Truffle. TASTyTruffle performs tail duplication at the AST level to generate guarded versions of polymorphic functions that are then specialized by the Graal compiler [13].

Our work is related to occurrence typing that is in particular used in Racket [33]. Occurrence typing is a type system that allows a context-sensitive refinement of variable types during static analysis, for instance by typing a variable differently in each branch of a conditional statement. In the context of our work, occurrence typing synergizes well with tail duplication to further propagate context-sensitive type information.

Partial function inlining can also be done if SBBV is applied interprocedurally. We have not done this in the current work because it is tricky to extend the versioning contexts to track code pointers for function entry and return points. This will require future work.

Code optimization by duplication and specialization has been extensively studied and used for various programming languages [12, 3, 29, 4, 9]. Recently Flückiger *et al.* [15] optimized the compilation of R programs by function specialization. They show that this technique makes programs run $1.7\times$ faster on average. Subsequently a study by Mehta *et al.* [21] has used a mechanism that keeps multiple versions of a given function specialized for contexts encountered at runtime by JIT compilers. Specialized versions of a function are stored in an external repository, allowing switching between versions when de-optimization occurs and to reuse versions of functions across executions and programs. SBBV is related to these techniques but the granularity of cloning is finer as it clones basic blocks while all

those previous works clone whole function bodies. This enables SBBV to better control the code expansion and to spend the cloning budget on relevant specializations without falling back on a fully generic version or using de-optimization, which is not an option in an AOT context. This sort of fine optimization tuning is out of reach for techniques that specialize at the level of whole function bodies.

6 Conclusion

Previous work has shown that the lazy variant of Basic Block Versioning (BBV) is effective in practice for optimizing dynamic checks in JIT compilers for dynamic languages [5, 27, 7, 8]. The *static* BBV (SBBV) approach that we have described in this paper is a variant of BBV that determines, through a fix-point program analysis, a set of basic block versions that are appropriate for the program and that covers all possible contexts without exceeding some versioning limit. This gives control over the code bloat induced by the multiple specializations of individual basic blocks in a way that avoids falling back on an unoptimized generic version of the basic block when the versioning limit is reached. SBBV is thus particularly interesting for use in AOT compilers and consequently it does not suffer from a warmup time.

At the core of the SBBV algorithm is a heuristic to drive the merging of previously generated versions to keep the number of versions within the allowed limit. We have shown through experiments that even a simple merge heuristic removes dynamic checks effectively in practice.

As a second contribution, we have shown in this paper how to extend the BBV approach to implement optimizations that go beyond the elimination of dynamic type checks. We have shown how to use it to remove integer overflow checks and bound checks effectively. By doing so, we have shown that the BBV approach can be viewed as a general programming analysis methodology that can be used to implement various optimizations that otherwise are implemented in isolation using dedicated techniques.

References

- 1 R7RS benchmarks. <https://github.com/ecraven/r7rs-benchmarks>, April 2024.
- 2 David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994. doi:10.1145/197405.197406.
- 3 Jeff Bezanson, Jiahao Chen, Benjamin Chung, Stefan Karpinski, Viral B. Shah, Jan Vitek, and Lionel Zoubitzky. Julia: dynamism and performance reconciled by design. *Proc. ACM Program. Lang.*, 2(OOPSLA):120:1–120:23, 2018. doi:10.1145/3276490.
- 4 Craig Chambers and David M. Ungar. Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In Richard L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN’89 Conference on Programming Language Design and Implementation (PLDI), Portland, Oregon, USA, June 21-23, 1989*, pages 146–160. ACM, 1989. doi:10.1145/73141.74831.
- 5 Maxime Chevalier-Boisvert and Marc Feeley. Simple and effective type check removal through lazy basic block versioning. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, volume 37 of *LIPICS*, pages 101–123. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPICS.ECOOP.2015.101.
- 6 Maxime Chevalier-Boisvert and Marc Feeley. Interprocedural type specialization of JavaScript programs without type analysis. 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 7:1–7:24. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPICS.ECOOP.2016.7.

28:26 Static Basic Block Versioning

- 7 Maxime Chevalier-Boisvert, Noah Gibbs, Jean Boussier, Si Xing (Alan) Wu, Aaron Patterson, Kevin Newton, and John Hawthorn. *YJIT: a basic block versioning JIT compiler for CRuby*, pages 25–32. ACM, 2021. doi:10.1145/3486606.3486781.
- 8 Maxime Chevalier-Boisvert, Takashi Kokubun, Noah Gibbs, Si Xing (Alan) Wu, Aaron Patterson, and Jemma Issroff. Evaluating YJIT’s performance in a production context: a pragmatic approach. In Rodrigo Bruno and Eliot Moss, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, MPLR 2023, Cascais, Portugal, 22 October 2023*, pages 20–33. ACM, 2023. doi:10.1145/3617651.3622982.
- 9 Keith D. Cooper, Mary W. Hall, and Ken Kennedy. A methodology for procedure cloning. *Comput. Lang.*, 19(2):105–117, 1993. doi:10.1016/0096-0551(93)90005-L.
- 10 Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977. doi:10.1145/512950.512973.
- 11 Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of generalized type unions. In David B. Wortman, editor, *Proceedings of an ACM Conference on Language Design for Reliable Software (LDRS), Raleigh, North Carolina, USA, March 28-30, 1977*, pages 77–94. ACM, 1977. doi:10.1145/800022.808314.
- 12 Iulian Dragos and Martin Odersky. Compiling generics through user-directed type specialization. In Ian Rogers, editor, *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, ICOOOLPS 2009, Genova, Italy, July 6, 2009*, pages 42–47. ACM, 2009. doi:10.1145/1565824.1565830.
- 13 Matt D’Souza, James You, Ondrej Lhoták, and Aleksandar Prokopec. TASTyTruffle: Just-in-time specialization of parametric polymorphism. *Proc. ACM Program. Lang.*, 7(OOPSLA2):1561–1588, 2023. doi:10.1145/3622853.
- 14 Shimon Even and Yossi Shiloach. An on-line edge-deletion problem. *J. ACM*, 28(1):1–4, 1981. doi:10.1145/322234.322235.
- 15 Olivier Flückiger, Guido Chari, Ming-Ho Yee, Jan Jecmen, Jakob Hain, and Jan Vitek. Contextual dispatch for function specialization. *Proc. ACM Program. Lang.*, 4(OOPSLA):220:1–220:24, 2020. doi:10.1145/3428288.
- 16 ECMA International. Standard ECMA-262 - ECMAScript language specification, June 2015. 6th edition. URL: <http://www.ecma-international.org/ecma-262/6.0/>.
- 17 Richard Kelsey, William D. Clinger, and Jonathan Rees. Revised⁵ report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998. doi:10.1145/290229.290234.
- 18 David Leopoldseder, Lukas Stadler, Thomas Würthinger, Josef Eisl, Doug Simon, and Hanspeter Mössenböck. Dominance-based duplication simulation (DBDS): code duplication to enable compiler optimizations. In Jens Knoop, Markus Schordan, Teresa Johnson, and Michael F. P. O’Boyle, editors, *Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, Vösendorf / Vienna, Austria, February 24-28, 2018*, pages 126–137. ACM, 2018. doi:10.1145/3168811.
- 19 Manuel Serrano. Bigloo. <http://www-sop.inria.fr/indes/fp/Bigloo/>, 2024.
- 20 Marc Feeley. Gambit. <https://gambitscheme.org>, 2024.
- 21 Meetesh Kalpesh Mehta, Sebastián Krynski, Hugo Musso Gualandi, Manas Thakur, and Jan Vitek. Reusing just-in-time compiled code. *Proc. ACM Program. Lang.*, 7(OOPSLA2):1176–1197, 2023. doi:10.1145/3622839.
- 22 Olivier Melançon, Marc Feeley, and Manuel Serrano. An executable semantics for faster development of optimizing Python compilers. In João Saraiva, Thomas Degueule, and Elizabeth Scott, editors, *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2023, Cascais, Portugal, October 23-24, 2023*, pages 15–28. ACM, 2023. doi:10.1145/3623476.3623529.

- 23 Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. *Introduction to Interval Analysis*. SIAM, 2009. doi:10.1137/1.9780898717716.
- 24 Frank Mueller and David B. Whalley. Avoiding conditional branches by code replication. In David W. Wall, editor, *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI), La Jolla, California, USA, June 18-21, 1995*, pages 56–66. ACM, 1995. doi:10.1145/207110.207116.
- 25 PLT Inc. Racket. <https://racket-lang.org/>, 2024.
- 26 R. Kent Dybvig. Chez Scheme. <https://www.scheme.com/>, 2024.
- 27 Baptiste Saleil and Marc Feeley. Interprocedural specialization of higher-order dynamic languages without static analysis. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of *LIPICS*, pages 23:1–23:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICS.ECOOP.2017.23.
- 28 Baptiste Saleil and Marc Feeley. Building JIT compilers for dynamic languages with low development effort. In Stephen Kell and Stefan Marr, editors, *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages, VMIL@SPLASH 2018, Boston, MA, USA, November 4, 2018*, pages 36–46. ACM, 2018. doi:10.1145/3281287.3281294.
- 29 Manuel Serrano. JavaScript AOT compilation. In Tim Felgentreff, editor, *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages, DLS 2018, Boston, MA, USA, November 6, 2018*, pages 50–63. ACM, 2018. doi:10.1145/3276945.3276950.
- 30 Manuel Serrano. Of JavaScript AOT compilation performance. *Proc. ACM Program. Lang.*, 5(ICFP):1–30, 2021. doi:10.1145/3473575.
- 31 Manuel Serrano and Marc Feeley. Property caches revisited. In José Nelson Amaral and Milind Kulkarni, editors, *Proceedings of the 28th International Conference on Compiler Construction, CC 2019, Washington, DC, USA, February 16-17, 2019*, pages 99–110. ACM, 2019. doi:10.1145/3302516.3307344.
- 32 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 395–406. ACM, 2008. doi:10.1145/1328438.1328486.
- 33 Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In Paul Hudak and Stephanie Weirich, editors, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 117–128. ACM, 2010. doi:10.1145/1863543.1863561.

Generalizing Shape Analysis with Gradual Types

Zeina Migeed  

University of California, Los Angeles (UCLA), CA, USA

James Reed 

Fireworks AI, Redwood City, CA, USA

Jason Ansel  

Meta, Menlo Park, CA, USA

Jens Palsberg  

University of California, Los Angeles (UCLA), CA, USA

Abstract

Tensors are multi-dimensional data structures that can represent the data processed by machine learning tasks. Tensor programs tend to be short and readable, and they can leverage libraries and frameworks such as TensorFlow and PyTorch, as well as modern hardware such as GPUs and TPUs. However, tensor programs also tend to obscure shape information, which can cause shape errors that are difficult to find. Such shape errors can be avoided by a combination of shape annotations and shape analysis, but such annotations are burdensome to come up with manually.

In this paper, we use gradual typing to reduce the barrier of entry. Gradual typing offers a way to incrementally introduce type annotations into programs. From there, we focus on tool support for *type migration*, which is a concept that closely models code-annotation tasks and allows us to do shape reasoning and utilize it for different purposes. We develop a comprehensive gradual typing theory to reason about tensor shapes. We then ask three fundamental questions about a gradually typed tensor program. (1) Does the program have a static migration? (2) Given a program and some arithmetic constraints on shapes, can we migrate the program according to the constraints? (3) Can we eliminate branches that depend on shapes? We develop novel tools to address the three problems. For the third problem, there are currently two PyTorch tools that aim to eliminate branches. They do so by eliminating them for just a single input. Our tool is the first to eliminate branches for an infinite class of inputs, using static shape information. Our tools help prevent bugs, alleviate the burden on the programmer of annotating the program, and improves the process of program transformation.

2012 ACM Subject Classification Software and its engineering → Software notations and tools

Keywords and phrases Tensor Shapes, Gradual Types, Migration

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.29

Related Version *Full Version:* <https://web.cs.ucla.edu/~palsberg/paper/ecoop24.pdf>

Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact):* <https://doi.org/10.4230/DARTS.10.2.14>

Acknowledgements We thank Akshay Utture, Micky Abir and Shuyang Liu for helpful comments.

1 Introduction

Multidimensional data structures are a common abstraction in modern machine learning frameworks such as PyTorch [13], TensorFlow [1], and JAX [5]. A significant portion of programs written using these frameworks involve transformations on tensors. Tensors in this setting are *n*-dimensional arrays. A tensor is characterized by its *rank* and *shape*. The *rank* is the number of dimensions. For example, a matrix is two-dimensional; hence it is a rank-2

 © Zeina Migeed, James Reed, Jason Ansel, and Jens Palsberg;
licensed under Creative Commons License CC-BY 4.0
38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 29; pp. 29:1–29:28



29:2 Generalizing Shape Analysis with Gradual Types

tensor. The *shape* captures the lengths of all axes of the tensor. For example, in a 2×3 matrix, the length of the first axis is 2 and the length of the second axis is 3; hence its shape is $(2, 3)$.

Programming with tensors provides the programmer with high level and easy to understand constructs. Furthermore, tensors can utilize modern hardware such as GPUs and TPUs for parallelization. For those reasons, programming with tensors is preferred over programming with scalars and nested loops.

Tensors in programming languages present the challenge that their shapes are hard to track. Modern machine learning frameworks support a plethora of operations on tensors, with complex shape rules. Addition for example, typically supports *broadcasting*, which is a mechanism that allows us to add tensors of different shapes, which is not intuitive. Complex shape rules make shapes hard to determine in programs, because shape information rarely explicitly appears in them. As a result, shape errors occur frequently [31].

When not caught statically, shape errors will appear at runtime, which is undesirable because we would only know about the error when the wrong operation is finally invoked on concrete runtime values. Tensor computations are costly and a program may take a long time to run before finally crashing with an error. Additionally, some shape errors occur only for specific input shapes.

The ability to reason about shapes is useful in various contexts in the machine learning area. It can prevent programmers from making mistakes and since programmers routinely transform machine learning programs [17], shape reasoning can also help program transformation tools to make valid program transformations because program transformations may depend on shape information.

Users often add asserts or comments to help them reason about shapes. These tasks have a high cognitive load on users, especially when they are dealing with complex tensor operations. Shape asserts present even further challenges; they can manifest in the form of branches on program shapes. We observed this pattern on various transformer benchmarks [30]. Thus, in that pattern, the result of a branch depends on the shape of the program input, so the branch result can vary over different inputs. In machine learning programs, branches can be undesirable because they limit the back-ends a program can be run on, such backends include TensorRT and XLA. The reason control-flow is undesirable is it complicates fix-point analysis, particularly in shape propagation [17]. In practice, various tools handle this challenge in different ways. Some tools reject such programs entirely while other tools run the program on a single input to eliminate branches. Running a program on a single input means that branch elimination is correct for just one input, which is an unsatisfactory solution.

Aiming to prevent the need for ad-hoc shape asserts, entire systems have been build to detect shape errors such as [15] and [24]. However, these systems are too specific. They lack a general theoretical foundation that enables their solution to be adapted to a variety of contexts, including incorporating their logic into compilers and program transformation tools.

A fundamental approach towards shape analysis is designing a type system that supports reasoning about shapes. In that approach, shapes are type annotations. Traditionally, types have been used to solve similar problems in the area of programming languages. A fully static type system with tensor shapes [20] has limitations. First, a static type system may need to be elaborate in order to capture the complexities of machine learning programs, which are typically written in permissive languages such as Python. As a result, refinement or polymorphic types may be needed. Second, a static type system has a high barrier of entry

because it requires the user to come up with non-trivial type annotations in advance. Third, many machine learning programs are in Python, so they are usually only partially typed. Therefore, fully typed programs are not readily available, which prevents this approach from being backwards-compatible.

A common way to circumvent the requirement of having fully typed programs is to use gradual types. In a gradually typed system, type annotations are not needed for the program to compile, when a compiler does type erasure. However, for a gradually typed system to be widely usable, it should enable principled yet practical tool support. Previous work such as [9] designed a gradually typed system for shapes but it is so powerful that practical, elaborate tool support may be hard to obtain. *We believe that the key to shape analysis with gradual types is to balance between (1) the expressiveness of a gradually typed system and (2) the ease of tool support in that system.*

We show that gradual types can help us tackle shape-related problems in a principled and unified way. We introduce a gradual typing system that reasons about shapes and enables tool support.

We distill the challenge of shape analysis into three key problems that we can ask of every gradually typed tensor program, and we introduce a general theory to solve all of them:

- Q(1): *Static migration*: Does the program have a static migration?
- Q(2): *Migration under arithmetic constraints*: Given a program and some arithmetic constraints on shapes, can we migrate the program according to the constraints?
- Q(3): *Branch elimination*: Can we eliminate branches that depend on shapes?

We use PyTorch as the setting for our tool design and evaluation, though our approach is more generally applicable. For Q(1) and Q(2), PyTorch does not currently have any comparable tools, so our tools for those challenges do something new in the PyTorch setting.

For Q(3), we incorporate our shape reasoning into two existing PyTorch tools that aim to eliminate branches from PyTorch programs. After augmenting both tools with our logic, we are able to improve the performance and accuracy of both tools as we will describe below. Our contributions can be summarized as follows:

1. A gradually typed tensor calculus that satisfies static gradual criteria [23].
2. A formal characterization of Q(1), Q(2) and Q(3) and their solutions.
3. A demonstration of how our approach works for Q(1) and Q(2) on four benchmarks.
4. For Q(3), a comparison on six benchmarks, against HuggingFace Tracer (**HFTtracer**) [30], a PyTorch tool. **HFTtracer** eliminates all branches based on a single input, while we eliminate all branches based on infinite classes of inputs. We use constraints to represent infinite classes of inputs.
5. For Q(3), a comparison on five benchmarks against **TorchDynamo** [2], a PyTorch tool. **TorchDynamo** eliminates 0% of the branches in these benchmarks, while we eliminate branches by 40% to 100% on infinite classes of inputs.

The full version has Appendices A–F with definitions and proofs.

2 Three Migration Problems

In this section, we introduce our type system informally, and we postpone the formal details to Section 3. A tensor type in our system is of the form `TensorType(d_1, \dots, d_n)` where d_1, \dots, d_n are dimensions.

Every gradually typed system has a type `Dyn`, which represents the absence of static type information. In our system, `Dyn` can appear as a dimension, in which case the dimension is unknown. `Dyn` can also appear as a tensor annotation, in which case even the rank of the tensor is unknown.

In a gradual type system, a precision relation refers to the replacement of some of the occurrences of `Dyn` with static types. `Dyn` is the least precise type because it contains no type information. `TensorType(1, 2, 3)` and `TensorType(1, 2)` are unrelated by the precision relation because we cannot go from one type to another by replacing `Dyn` occurrences with more informative types, while `TensorType(Dyn, 2)` is less precise than `TensorType(1, 2)` because we can replace the `Dyn` in

`TensorType(Dyn, 2)` with 1 to get `TensorType(1, 2)`. This relation extends to programs. Program *A* is less precise than program *B* if we can replace some occurrences of `Dyn` in program *A* to get to program *B*. Intuitively, program *B* is more static than program *A*. Precision gives rise to the *migration space* [12]. Given a well-typed program *P*, its migration space is the set of well-typed programs that are at least as precise as *P*.

Intuitively, the migration space captures all ways of annotating a gradually typed program more precisely. Those possibilities form a partially ordered set, and our goal is to help the programmer find the migration paths they are looking for. With that in mind, let us look at examples of how reasoning about the migration space is beneficial for solving key problems about the shapes in a gradually typed program. Specifically, in Section 2, we will see two examples about Q(1) and Q(2) respectively, and in Section 2, we will see an example about Q(3).

For an example of static migration, consider Listing 1 which has a type error.

```

1 class ConvExample(torch.nn.Module):
2     def __init__(self):
3         super(BasicBlock, self).__init__()
4         self.conv1 = torch.nn.Conv2d(in_channels=2, ...)
5         self.conv2 = torch.nn.Conv2d(in_channels=4, ...)
6
7     def forward(self, x: TensorType([Dyn, Dyn])):
8         self.conv1(x)
9         return self.conv2(x)

```

■ Listing 1 Ill-typed convolution.

In line 7, `x` is annotated with `TensorType([Dyn, Dyn])`. This is a typical gradual typing annotation which indicates that `x` is a rank-2 tensor. The annotation does not specify what the dimensions are. In line 8, we are applying a convolution to `x`. Intuitively, convolution is a variant of matrix multiplication; neural networks use it to extract features from images. According to PyTorch’s documentation, for the convolution to succeed, `x` cannot be rank-2. Thus, the type error stems from a wrong type annotation. The migration space of this program can easily inform us that the program is ill-typed, because the space will be empty. The reason for that is that the migration space of a well-typed program should contain at least one element, which is the program itself. A tool that can reason about the migration space can easily catch this bug in a single step.

Let us fix this bug by replacing the wrong type annotation with a correct one. In Listing 2, we change `x`’s annotation from a rank-2 annotation to a rank-4 annotation: `TensorType([Dyn, Dyn, Dyn, Dyn])`, which is correct. This program compiles, but it contains a more subtle bug. Let us look closely at the code to understand why.

In line 4, we initialize a field, `self.conv1`, representing a convolution, `torch.nn.Conv2d`, which takes various parameters. The parameter that’s relevant to our point is called `in_channels` and it is set to 2. In line 5, we are initializing another field, `self.conv2`, but this time, we set the `in_channels` to 4. In line 7, we have a function that takes a variable `x` and calls both convolutions on it in lines 8 and 9. To understand why this program contains a bug, we must ask: *how does the value of `in_channels` relate to `x`’s shape?* PyTorch’s

documentation [14] states that in the simplest case, the input to a convolution has the shape $(N, \text{in_channels}, H, W)$. Indeed, in line 7, `x` is annotated with `TensorType([Dyn, Dyn, Dyn, Dyn])`, a typical gradual typing annotation indicating that `x` is a rank-4 tensor. The annotation does not state what the dimensions are, but it is still consistent with the shape stated in the documentation. Notice however that `x`'s second dimension should match the value of `in_channels`, while we have two values for `in_channels` that do not match. This mismatch will cause the program to crash if it ever receives any input, but not before. Our key questions can help us discover the bug statically across all inputs.

```

1 class ConvExample(torch.nn.Module):
2     def __init__(self):
3         super(BasicBlock, self).__init__()
4         self.conv1 = torch.nn.Conv2d(in_channels=2, ...)
5         self.conv2 = torch.nn.Conv2d(in_channels=4, ...)
6
7     def forward(self, x: TensorType([Dyn, Dyn, Dyn, Dyn])):
8         self.conv1(x)
9         return self.conv2(x)

```

■ Listing 2 Gradually typed convolution.

By determining whether we can replace all the `Dyn` dimensions with numbers (which is the answer to Q(1) from our key questions), we can discover that it is impossible to assign a number to the second dimension of `x` and thus detect the error before running the program. More generally, the absence of a static typing may reveal that a program cannot run successfully on any input.

How can we benefit from the migration space to answer Q(1) and thus detect that this program cannot be statically typed? The migration space for this program contains programs where `x` is annotated to be a rank-4 tensor. A tool that can reason about the migration space can then take an extra constraint on the second dimension of `x`. The constraint should say that the second dimension must be a number. This constraint will narrow down the migration space to an empty set. The reason is that there is no such well-typed program. Therefore, we can conclude that the program cannot be statically typed because the second dimension cannot be assigned a number.

Let us fix the bug. One way to fix the bug is by removing `self.conv1` from the program. We get the program in Listing 3.

```

1 class ConvExample(torch.nn.Module):
2     def __init__(self):
3         super(BasicBlock, self).__init__()
4         self.conv2 = torch.nn.Conv2d(in_channels=4, ...)
5
6     def forward(self, x: TensorType([Dyn, Dyn, Dyn, Dyn])):
7         return self.conv2(x)

```

■ Listing 3 Gradually typed convolution.

The program can run to completion and there can be various correct ways to annotate it. The current annotation for the variable `x` is that it is a tensor with four dimensions, but each dimension is denoted by `Dyn`, so the values of the dimensions are unknown. Suppose we want to specify constraints on those dimensions and determine if there are valid migrations that satisfy those constraints. This would be useful, not just for the user, but for compilers, since they can use those constraints to optimize for resources.

We can require some of the dimensions of `x` to be static and then provide arithmetic constraints on each of them. In this example, let us require all dimensions to be static. A tool can accept four constraints indicating this requirement. Then it can accept constraints

29:6 Generalizing Shape Analysis with Gradual Types

that specify ranges on those dimensions. For example, the first dimension could be between 5 and 20. The second dimension can only have one possible value, which is 4. So it is enough to have a constraint requiring that dimension to be a number. The third dimension could also be between 5 and 20, while the fourth dimension could be between 2 and 10.

By giving these constraints as input to a tool, we are constraining the space to only the subspace that satisfies the constraints. A tool may find that this subspace indeed contains programs and outputs one of them. As a result, we may get the program in Listing 4. As shown, `x` has now been statically annotated with `TensorType([19, 4, 19, 9])`.

```
1 class ConvExample(torch.nn.Module):
2     def __init__(self):
3         super().__init__()
4         self.conv2 = torch.nn.Conv2d(in_channels=4, ...)
5     def forward(self, x: TensorType([19, 4, 19, 9])):
6         return self.conv2(x)
```

■ Listing 4 Statically typed convolution.

```
1 class ConvControlFlow(torch.nn.Module):
2     def __init__(self):
3         super().__init__()
4         self.conv = torch.nn.Conv2d(
5             in_channels=512, out_channels=512, kernel_size=3)
6
7     def forward(self, x: TensorType([Dyn, Dyn, Dyn, Dyn])):
8         if self.conv(x).dim() == 4:
9             return torch.relu(x)
10        else:
11            return torch.nn.Dropout(x)
```

■ Listing 5 Branch elimination.

The program in Listing 5 can run to completion, and interestingly it contains control-flow in the form of a branch. We want to eliminate this branch. We refer to eliminating branches from a program by the term *branch elimination*. Eliminating branches enables programs to run on back-ends where branches are undesirable. For example, `HFTtracer` runs a program on a single input and computes the result of the branch and eliminates it accordingly. While the result of a branch could be fixed for all program inputs, the result may also vary. Thus, running a program on just a single input to eliminate a branch yields unsatisfactory branch elimination. We enable better branch elimination by finding all inputs for which a branch evaluates to a given result by reasoning about the program statically. We provide a mechanism to denote the set of inputs for which a branch evaluates to the given result. Notice that we reason about the static information given. Thus, if a variable has type `Dyn`, we optimistically assume that the program is well-typed and that the value for that variable will have the appropriate type at runtime.

The program in Listing 5 contains a condition that depends on shape information. This is a common situation, where ad-hoc shape-checks are inserted in a program to reason about its shapes. Line 8 has function that takes a variable `x` and applies a convolution to it, with `self.conv(x)`, and a condition that checks if the rank of `self.conv(x)` is 4. Since `x` is annotated as a rank-4 tensor on line 7, and convolution preserves the rank, `self.conv(x)` must also be rank-4. So the condition must always be true under the information given by `x`'s type annotation. We should be able to prove that the condition in line 8 always returns true without receiving any input for the program, by inspecting all the valid types that the program could possibly have. The migration space is useful for this analysis because it captures all possible, valid type annotations for a program.

Thus, under the convolution type rules, if `self.conv(x).dim() == 4` evaluates to true, then `x` is also rank-4, which is consistent with `x`'s current annotation.

In contrast, if `self.conv(x).dim() == 4` evaluates to false, i.e `self.conv(x).dim() != 4` is true, then this means that `x` is not rank-4. However, the migration space of a program can never include inconsistent ranks for a variable. Therefore, it is impossible to have `self.conv(x).dim() != 4`, while also having that `x` is rank-4. A tool that reasons about the migration space as well as arbitrary predicates can make this conclusion. In this example, we can make a definitive conclusion about the result of this condition and we can re-write our program accordingly, as shown in Listing 6. We will expand on and formalize this idea in Section 5. In particular, we will detail how we reason about the migration space in the presence of branches, and explain why our approach works.

```

1 class ConvControlFlow(torch.nn.Module):
2     def __init__(self):
3         super().__init__()
4         self.conv = torch.nn.Conv2d(
5             in_channels=512, out_channels=512, kernel_size=3)
6
7     def forward(self, x: TensorType([Dyn, Dyn, Dyn, Dyn])):
8         return torch.relu(x)

```

 **Listing 6** Branch elimination.

3 The Gradual Tensor Calculus

In this section, we describe our design choices, core calculus, and type system, and we prove that our type system satisfy gradual typing criteria.

Our design choices are guided by enabling four key requirements: (1) modularity and backwards compatibility, (2) tool support, (3) expressiveness, and (4) minimality of our language. We have made these four choices in the context of tool support for PyTorch, but they can be extended to other frameworks. Here, we outline those design choices.

First, we require our system to support *modularity and backwards compatibility* for programs. A gradually typed system suits our needs because it supports partial type annotations. One of the implications of this support is that gradually typed programs can compile with any amount of type annotations. In a gradually typed system, a missing type is represented by the `Dyn` type.

The `Dyn` type can sometimes be assigned to a variable that has been used in different parts of the program with different, possibly inconsistent types. This type is useful when the underlying static type system is not flexible enough to fully type that program. For example, we may have a program that takes a batch of images with a dynamic batch size, as well as dynamic sizes, but with a fixed number of channels. In this case, a possible type would be `TensorType(Dyn, 3, Dyn, Dyn)`, which indicates a batch of images, where the batch size is dynamic and the sizes are dynamic but the number of channels, which is 3, is fixed. Another example is that a variable could be assigned a rank-2 tensor at one point in the program, then a rank-3 tensor at a different point. A suitable type for that variable could simply be `Dyn`. In both examples, if we did not have the `Dyn` type, we would need more complex annotations. The `Dyn` type allows the gradual type checker to admit programs statically, and determine how to handle variables with `Dyn` types at runtime. The flexibility of gradual types stems from the consistency relation, which is symmetric and reflexive but not transitive. This relation allows a gradual type checker to statically admit programs in the absence of type information.

$$\begin{array}{l}
\text{(Program)} \quad p ::= \text{decl}^* \text{return } e \\
\text{(Declaration)} \quad \text{decl} ::= x : \tau \\
\text{(Expression)} \quad e ::= x \mid \text{reshape}(e, \tau) \mid \text{Conv2D}(c_{in}, c_{out}, \kappa, e) \mid \text{add}(e_1, e_2) \\
\text{(Integer Tuple)} \quad \kappa ::= (c^*) \\
\text{(Const)} \quad c ::= \langle \text{Nat} \rangle \\
\text{(Tensor Type)} \quad t, \tau ::= \text{Dyn} \mid \text{TensorType}([d_1, \dots, d_n]) \\
\text{(Static Tensor Type)} \quad S, T ::= \text{TensorType}([D_1, \dots, D_n]) \\
\text{(Dimension Type)} \quad d, \sigma ::= \text{Dyn} \mid D \\
\text{(Dimension)} \quad U, D ::= \langle \text{Nat} \rangle
\end{array}$$

$$\frac{x \notin \text{dom}(\Sigma)}{\Sigma, x \rightarrow^* \Sigma, 0, 1} (\text{Var Fail}) \qquad \frac{x : R \in \Sigma}{\Sigma, x \rightarrow^* \Sigma, R, 0} (\text{Var})$$

$$\frac{\Sigma, e \rightarrow^* \Sigma, R, 1}{\Sigma, \text{reshape}(e, \text{TensorType}(d_1, \dots, d_n)) \rightarrow^* \Sigma, R, 1} (\text{Reshape Fail})$$

$$\frac{\Sigma, e \rightarrow^* \Sigma, R, 1}{\Sigma, \text{Conv2D}(c_{in}, c_{out}, \kappa, e) \rightarrow^* \Sigma, R, 1} (\text{Conv2D Fail})$$

$$\frac{\Sigma, e_1 \rightarrow^* \Sigma, R_1, 1 \vee \Sigma, e_2 \rightarrow^* \Sigma, R_2, 1}{\Sigma, \text{add}(e_1, e_2) \rightarrow^* \Sigma, R_2, 1} (\text{Add Fail})$$

$$\frac{\Sigma, e \rightarrow^* \sigma, R, 0}{\Sigma, \text{reshape}(e, \text{TensorType}(d_1, \dots, d_n)) \rightarrow^* \Sigma, \text{RESHAPE}(R, (d_1, \dots, d_n))} (\text{Reshape})$$

$$\frac{\Sigma, e \rightarrow^* \Sigma, R, 0}{\Sigma, \text{Conv2D}(c_{in}, c_{out}, \kappa, e) \rightarrow^* \Sigma, \text{CONV2D}(c_{in}, c_{out}, \kappa, R)} (\text{Conv})$$

$$\frac{\Sigma, e_1 \rightarrow^* \Sigma, R_1, 0 \quad \Sigma, e_2 \rightarrow^* \Sigma, R_2, 0}{\Sigma, \text{add}(e_1, e_2) \rightarrow^* \Sigma, \text{ADD}(R_1, R_2)} (\text{Add})$$

 **Figure 1** Gradual tensor calculus, syntax and semantics.

Second, we require *tool support*. We design a simple type system for a core language to enable us to define and solve problems for tool support in a tractable way. Tool support is tractable because we define type migration syntactically. We base our approach on capturing the migration space by extending the constraint-based approach of [12] to solve our three key questions.

Third, we require our system to be *expressive* enough to capture non-trivial programs. Our type system is more expressive than PyTorch’s existing type-system, which does not reason about dimensions. Our language consists of a set of declarations followed by an expression. This structure is a convenient representation for the PyTorch neural network models we encountered, which mainly consisted of a function which takes a set of parameters. In the function body are tensor operations applied on those parameters. This calculus structure is inspired by the calculus from [18]. Rink highlighted that many DSLs can be mapped to their language. Besides adapting the structure of that calculus, we choose three core operations that present different challenges for tool support, and then extend our support to 50 PyTorch operations.

Fourth, we require our language to be *minimal* so we can focus on our core problems. First, we do not introduce branches to our core grammar since, in practice, all tools on which we ran our experiments either do not accept programs with branches or aim to eliminate branches. As [17] noted, many non-trivial tensor programs do not contain branches or statements. In Section 5 we extend the core language with branches and we show how to eliminate them.

Second, we do not consider runtime checks to support gradual types. Those checks are often a bottleneck for the performance of gradually typed programs [25, 8]. There has been extensive research to alleviate performance issues by weakening these checks. As shown by [7], the notion of soundness in gradual types is not an all-or-nothing concept. [7] discuss three notions of soundness at different levels of strength and how they relate to performance: higher-order embedding of [26], first-order embedding, as seen in Reticulated Python [28] and erasure embedding, as seen in TypeScript [4]. Similar to [18] and [17], we observe that a language free from higher-order constructs represents a large subset of programs that are written in the machine learning area. As such, runtime errors are not as interesting when compared to those that arise in languages with constructs such as branches and lambda-abstraction. Furthermore, runtime checks impose a computation cost on already costly tensor computations. A key goal of tensor programming is high performance so adding run-time checks seems undesirable. Thus, we leave out runtime aspects in this paper.

Figure 1 shows our core calculus. A program consists of a list of declarations followed by a return statement that evaluates an expression. We use ϵ to denote the empty list of declarations. The program takes its input via those declarations. The dynamic type is denoted by `Dyn`. A dimension can be `Dyn`, and a tensor can also be `Dyn`. A tensor is denoted by the constructor `TensorType($\sigma_1, \dots, \sigma_n$)` where $\sigma_1, \dots, \sigma_n$ are dimensions. However, if we denote a dimension by `U` or `D`, it means the dimension is a number and cannot be `Dyn`. Our language has four kinds of expressions. A variable x refers to one of the declared variables. The expression `add(e_1, e_2)` adds two tensors e_1 and e_2 . The expression `reshape(e, τ)` takes an expression e and a shape τ and reshapes e to a new tensor of shape τ if possible. Reshaping can be thought of as a re-arrangement of a tensor’s elements. That requires the initial tensor to have the same number of elements as the reshaped tensor. We require that τ can have a maximum of one `Dyn` dimension. Finally the expression `Conv2D($c_{in}, c_{out}, \kappa, e$)` applies a convolution to e , given a number representing the input channel c_{in} , a number representing the output channel c_{out} , and a pair of numbers representing the kernel κ . For example, in Listing 2, we had `self.conv1(x)`, which in our calculus can be expressed as `Conv2D(2, 2, (2, 2), x)`. The full version of convolution in PyTorch has more parameters. We have accounted for those parameters in our implementation, but because they create no new problems for us, our quest for minimality led us to leaving them out.

The operational semantics in Figure 1 evaluates an expression in an environment Σ that maps each declared variable to a tensor constant. Specifically, if e is an expression, R is a tensor constant, and E an error state (0 for success, 1 for failure), then the judgment $\Sigma, e \rightarrow^* R, E$ means that e evaluates to R in error state E .

The semantics uses the helper functions `ADD`, `RESHAPE`, and `CONV2D` that each produces both a tensor constant and an error state. In Appendix C, we give full details of those functions and we state their key properties. Here we summarize what they do. The function `ADD` extracts shapes from T_1 and T_2 and pads them such that they match, and then checks if the tensors are broadcastable based on the updated shapes. If they are not broadcastable, it returns the empty tensor with $E = 1$. Otherwise, it expands the tensors T_1 and T_2 according to the broadcasting rules of PyTorch that we omit here. It initializes a resulting tensor with

29:10 Generalizing Shape Analysis with Gradual Types

Consistency

$$\begin{array}{lll} \tau \sim \tau \text{ (c-refl-t)} & d \sim d \text{ (c-refl-d)} & d \sim \text{Dyn } (d\text{-refl-dyn}) \\ \frac{t \sim \tau}{\tau \sim t} \text{ (c-sym-t)} & \frac{d \sim \sigma}{\sigma \sim d} \text{ (c-sym-d)} & \tau \sim \text{Dyn } (t\text{-refl-dyn}) \end{array}$$

$$\frac{\forall i \in \{1, \dots, n\} : d_i \sim d'_i}{\text{TensorType}(d_1, \dots, d_n) \sim \text{TensorType}(d'_1, \dots, d'_n)} \text{ (c-tensor)}$$

Type Precision

$$\frac{\tau \sqsubseteq \tau \text{ (refl-t)} \quad d \sqsubseteq d \text{ (c-refl-d)} \quad \text{Dyn} \sqsubseteq d \text{ (refl-dyn-1)} \quad \text{Dyn} \sqsubseteq \tau \text{ (refl-dyn-2)}}{\text{TensorType}(d_1, \dots, d_n) \sqsubseteq \text{TensorType}(d'_1, \dots, d'_n)} \text{ (p-tensor)}$$

Program and Expression Precision

$$\frac{\forall i \in \{1, \dots, n\} : \text{decl}'_i \sqsubseteq \text{decl}_i \quad e' \sqsubseteq e}{\text{decl}'_1, \dots, \text{decl}'_n \text{ return } e' \sqsubseteq \text{decl}_1, \dots, \text{decl}_n \text{ return } e} \text{ (p-prog)} \quad \frac{\tau' \sqsubseteq \tau}{x : \tau' \sqsubseteq x : \tau} \text{ (p-decl)}$$

$$e \sqsubseteq e \text{ (p-refl)}$$

Matching

$$\frac{\text{TensorType}(\tau_1, \dots, \tau_n) \triangleright^n \text{TensorType}(\tau_1, \dots, \tau_n)}{\text{Dyn} \triangleright^n \text{TensorType}(l) \text{ where } l = [\text{Dyn}, \dots, \text{Dyn}] \text{ and } |l| = n}$$

Static context formation

$$\frac{}{\epsilon \vdash \emptyset} \text{ (s-empty)} \quad \frac{\text{decl}^* \vdash \Gamma \quad x \notin \text{dom}(\Gamma)}{\text{decl}^* x : \tau \vdash \Gamma, x : \tau} \text{ (s-var)}$$

 **Figure 2** Auxiliary functions.

the broadcasted dimensions and perform element-wise addition between the broadcasted tensors and return that tensor with $E = 0$. The function RESHAPE performs dimension checks to ensure that reshaping is possible, returning the empty tensor and $E = 1$ if the checks fails. Otherwise, it performs reshaping and returns the reshaped tensor with $E = 0$. The function CONV2D extracts the dimensions of the input tensor I , as well the dimensions for the kernel κ and uses them to determine the size of the output tensor. It then performs convolution and populates the output tensor one element at a time and return the updated tensor along with $E = 0$.

The semantics satisfies the following theorem, which says that in an environment, an expression evaluates to a tensor but may end with failure.

► **Theorem 1.** $\forall \Sigma, e : \exists a \text{ tensor constant } R : \exists E \in \{0, 1\} : \Sigma, e \rightarrow^* R, E$.

Figure 2 contains gradual typing relations that are used in our gradual typechecking, as well as the static context formation rules. Those relations allow the typechecker to reason about the `Dyn` type. Matching, denoted by \triangleright , and consistency, denoted by \sim , are standard in gradual typing and are lifted from equality in the static counter part of the system. Matching and consistency are both weaker than equality because they account for absent type information. Thus, if some type information is missing, matching and consistency apply. Matching is a relation that pattern-matches two types. It is useful for arrow types

$$\begin{array}{c}
\frac{\text{decl}^* \vdash \Gamma \quad \Gamma \vdash e : \tau}{\vdash \text{decl}^* \text{return } e \text{ ok}} \text{ (ok-prog)} \quad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \text{ (t-var)} \\
\\
\frac{\Gamma \vdash e : \text{TensorType}(D_1, \dots, D_n) \quad \prod_1^n D_i = \prod_1^m U_i}{\Gamma \vdash \text{reshape}(e, \text{TensorType}(U_1, \dots, U_m)) : \text{TensorType}(U_1, \dots, U_m)} \text{ (t-reshape-s)} \\
\\
\frac{\Gamma \vdash e : \text{TensorType}(\sigma_1, \dots, \sigma_m)}{\prod_1^m \sigma_i \text{ mod } \prod_1^n d_i = 0 \vee \prod_1^n d_i \text{ mod } \prod_1^m \sigma_i = 0 \quad \forall d_i, \sigma_i \neq \text{Dyn} \text{ and}} \\
\text{Dyn occurs exactly once in } d_1, \dots, d_m, \sigma_1, \dots, \sigma_n, \text{ or} \\
\text{Dyn occurs more than once in } d_1, \dots, d_m, \\
\frac{\text{Dyn occurs more than once with at least one occurrence in } \delta \text{ and } \sigma_1, \dots, \sigma_m}{\Gamma \vdash \text{reshape}(e, \text{TensorType}(d_1, \dots, d_n)) : \text{TensorType}(d_1, \dots, d_n)} \text{ (t-reshape-g)} \\
\\
\frac{\Gamma \vdash e : \tau \text{ where either } \tau = \text{Dyn}, \text{ or } \tau = \text{TensorType}(\sigma_1 \dots \sigma_n) \text{ and}}{\Gamma \vdash \text{reshape}(e, \delta) : \delta} \text{ (t-reshape)} \\
\\
\frac{\Gamma \vdash e : t \quad t \triangleright^4 \text{TensorType}(\sigma_1, \sigma_2, \sigma_3, \sigma_4) \quad \tau = \text{calc-conv}(t, c_{out}, \kappa) \quad c_{in} \sim \sigma_2}{\Gamma \vdash \text{Conv2D}(c_{in}, c_{out}, \kappa, e) : \tau} \text{ (t-conv)} \\
\\
\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2 \quad (\tau_1, \tau_2) = \text{apply-broadcasting}(t_1, t_2) \quad \tau_1 \sim \tau_2}{\Gamma \vdash \text{add}(e_1, e_2) : \tau_1 \sqcup^* \tau_2} \text{ (t-add)}
\end{array}$$

■ **Figure 3** Type rules.

in traditional type systems. Specifically, an arrow type $t_1 \rightarrow t_2$ matches itself. Type Dyn matches $\text{Dyn} \rightarrow \text{Dyn}$. The ability to expand Dyn to become a function type $\text{Dyn} \rightarrow \text{Dyn}$ is valid in gradual types because it allows the system to optimistically consider the type Dyn to be $\text{Dyn} \rightarrow \text{Dyn}$. We have adapted this definition to our system. First, we annotated matching with a number n to denote the number of dimensions involved. So we have that $\text{TensorType}(\tau_1, \dots, \tau_n) \triangleright^n \text{TensorType}(\tau_1, \dots, \tau_n)$ because any type matches itself. Similar to how traditionally, $\text{Dyn} \triangleright \text{Dyn} \rightarrow \text{Dyn}$, we have that $\text{Dyn} \triangleright^n \text{TensorType}(\text{Dyn}, \dots, \text{Dyn})$, where $\text{Dyn}, \dots, \text{Dyn}$ are exactly n dimensions. Throughout this paper, we will only use matching with $i = 4$ so we may use matching as \triangleright instead of \triangleright^4 . Consistency is a symmetric, reflexive, and non-transitive relation that checks that two types are equal, up to the known parts of the types. For example, the type Dyn contains no information, so it is consistent with any type, while the dimensions 3 and 4 are inconsistent because they are unequal. Figure 2 contains the formal definitions for matching and consistency. The judgment $\text{decl}^* \vdash \Gamma$ says that from the declarations decl^* we get the environment Γ . We do static context formation with the rules (*s-empty*) and (*s-var*).

Figure 3 shows our type rules. We use shorthands that are defined in Appendix B. Let us go over each type rule in detail. *ok-prog* and *t-var* are standard.

t-reshape-s is the static type rule for reshape. It models that for reshape to succeed, the product of the dimensions of the input tensor shape must equal the product of dimensions of the desired shape. *t-reshape-g* assumes we have one missing dimension. Here we are modeling that PyTorch allows a programmer to leave one dimension as unknown (denoted by -1) because the system can deduce the dimension at runtime, see <https://pytorch.org/docs/>

29:12 Generalizing Shape Analysis with Gradual Types

[stable/generated/torch.reshape.html](#). We can still determine if reshaping is possible using the modulo operation instead of multiplication. In this approach, we admit a program if we cannot prove it is ill-typed statically. *t-reshape* admits the expression if too many dimensions are missing.

To maintain minimality, *t-conv* deals with only the rank-4 case of convolution. *t-conv* expects a rank-4 tensor, so it uses matching (\triangleright^4) to check the rank. Next, c_{in} should be equal to the second dimension of the input, so the rule uses a consistency (\sim) check. Since the output of a convolution should also be rank-4, then apply *calc-conv* which, given a rank-4 input and the convolution parameters, computes the dimensions of the output shape. If a dimension is *Dyn*, then the corresponding output dimension will also be *Dyn*.

Finally, *t-add* adds two dimensions. Unlike scalar addition, the types of the operands do not have to be consistent. The reason is that broadcasting may take place. Broadcasting is a mechanism that considers two tensors and matches their dimensions. Two tensors are broadcastable if the following rules hold:

1. Each tensor has at least one dimension
2. When iterating over the dimension sizes, starting at the trailing dimension, the dimension sizes must either be equal, one of them is 1, or one of them does not exist

That tensors involved in broadcasting do not actually get modified to represent the modified shapes. This implies that the input shapes are not always consistent. Instead, the broadcasted result is only reflected in the output of the operation. Therefore, we have defined *apply-broadcasting* to simulate broadcasting on the inputs and consider what the types for these inputs would be, if broadcasting was to actually modify the inputs. In a static type system, the types of the modified inputs should be equal for addition to succeed. In gradual types, the types of the modified inputs should be consistent because equality lifts to consistency. We accomplish these requirements in our type rule. In particular, *apply-broadcasting* takes care of broadcasting the dimensions. Suppose that we are adding a tensor of shape `TensorType(Dyn, 2, Dyn)` to a tensor of size `TensorType(1, 2, 2)`. Then the output must be `TensorType(Dyn, 2, 2)`. The reason is that the first *Dyn* could be any number as per the broadcasting rules. So we cannot assume its value. The last dimension; however, must be 2 according to the rules. We have that:

$$\begin{aligned} \text{apply-broadcasting}(\text{TensorType}(\text{Dyn}, 2, \text{Dyn}), \text{TensorType}(1, 2, 2)) = \\ (\text{TensorType}(\text{Dyn}, 2, \text{Dyn}), \text{TensorType}(\text{Dyn}, 2, 2)) \end{aligned}$$

After simulating broadcasting, we may proceed as if we are dealing with regular addition. In other words, we check that the modified dimensions are consistent and get the least upper bound: `TensorType(Dyn, 2, Dyn) \sqcup TensorType(Dyn, 2, 2) = TensorType(Dyn, 2, 2)`.

We will cover one last special case for addition. Simply applying the least upper bound to the modified input types of addition is not general enough to cover the following case. Suppose we are adding a tensor of shape *Dyn* to a tensor of shape `TensorType(1, 2)`, then we must output *Dyn* because the output type could be a range of possibilities. In this case, *apply-broadcasting* does not modify the types because the tensor of shape *Dyn* could range over many possibilities. We then apply our modified version of the least upper bound denoted by \sqcup^* , which behaves exactly like \sqcup except when one of the inputs is *Dyn*, where it returns *Dyn* to get that: `TensorType(1, 2) \sqcup^* Dyn = Dyn`.

We prove that our type system satisfies the static criteria from [23]. First, we prove the static gradual guarantee, which describes the structure of the migration space. Second, we prove the conservative extension theorem, which shows that our gradual calculus subsumes its static counter-part in Appendix A. This result is no coincidence: we first designed the

statically typed calculus in Appendix A and then we gradualized it according to [6]. We denote a well-typed program in the statically typed tensor calculus by $\vdash_{st} p : \text{ok}$. The full definitions and proofs can be found in Appendix D.

► **Theorem 2** (Monotonicity w.r.t precision). $\forall p, p' : \text{if } \vdash p : \text{ok} \wedge p' \sqsubseteq p \text{ then } \vdash p' : \text{ok}$.

► **Theorem 3** (Conservative Extension). *For all static p , we have: $\vdash_{st} p : \text{ok}$ iff $\vdash p : \text{ok}$*

4 The Migration Problem as a constraint satisfiability problem

A migration is a more static, well-typed version of a program. We can define that P' is a migration of P (which we write $P \leq P'$) iff $(P \sqsubseteq P' \wedge \vdash P' : \text{ok})$. Given P , we define the set of migrations of P : $Mig(P) = \{P' \mid P \leq P'\}$. Our goal is to use constraints to capture the migration space. Every solution to our constraints for a program must map to a corresponding migration for the same program. In other words, one satisfying assignment to the constraints results in one migration.

Our approach involves defining constraints whose solutions are order-isomorphic with the migration space. However, due to the arithmetic nature of our constraints, our solution procedure uses an SMT solver to find a satisfying assignment, which would equate to finding a migration. Later in this paper, we will show how to use this framework to answer our three key questions.

We have two grammars of constraints, see Figure 4: one for source constraints and one for target constraints. We will generate source constraints and then map them to target constraints (as explained in Appendix E), and finally process the target constraints by an SMT solver. Having two grammars is not strictly necessary, but it makes the constraint generation process more tractable and simplifies the presentation. We can view the source grammar as syntactic sugar for the target grammar.

Our source constraint grammar has fourteen forms of constraints, the most interesting of which we will introduce here. A precision constraint is of the form $\tau \sqsubseteq x$. Here, x indicates a type variable for the variable x from the program. Thus, x in the constraint $\tau \sqsubseteq x$ captures all types that are more precise than τ . Because we prioritize tractability of the migration space, we set the upper bound of tensor ranks to 4, via a constraint of the form $|[\![e]\!]| \leq 4$. We make this decision because all benchmarks we considered had only tensors with ranks that are upper-bounded by this number. We also have consistency constraints of the form $D \sim \delta, \langle e \rangle \sim \langle e \rangle$, matching constraints of the form $[\![e]\!] \triangleright \text{TensorType}(\delta_1, \delta_2, \delta_3, \delta_4)$, and least upper bound constraints of the form $\langle e \rangle \sqcup^* \langle e \rangle$. Those are gradual typing constraints that we use to faithfully model our gradual typing rules. Our constraint grammar also contains short-hands such as `can-reshape([\![e]\!], δ)` and `apply-broadcasting([\![e]\!], [\![e]\!])`. Those short-hands are good for representing the type rules as well. `can-reshape` expands to further constraints which evaluate to true if $[\![e]\!]$ can be reshaped to δ . Similarly, when expanded, `apply-broadcasting([\![e]\!], [\![e]\!])` captures all possible ways to broadcast two types.

In our target constraint grammar, we use n to range over integer constants. We use v as a meta variable that ranges over variables that, in turn, range over $\text{TensorType}(\text{list}(\zeta)) \cup \{\text{Dyn}\}$ and we use ζ as a meta variable that ranges over variables that range over $\text{IntConst} \cup \{\text{Dyn}\}$. This grammar is useful for our constraint resolution process. In particular, the first step of solving our constraints is to translate them to low-level constraints, drawn from our target grammar, before feeding them to an SMT solver.

Since our constraints involve gradual types, let us describe how we encoded types so that they can be understood by an SMT solver. Because we fixed the upper bound for tensor ranks to be 4, we chose to encode tensor types as uninterpreted functions, which means

$$\begin{aligned}
(\text{Source Constraints}) \quad \psi ::= & \quad \psi \wedge \psi \mid \psi \vee \psi \mid \text{True} \mid \llbracket x \rrbracket = x \mid \llbracket e \rrbracket = \tau \mid \tau \sqsubseteq x \mid \\
& \llbracket e \rrbracket \leq 4 \mid D \sim \delta \mid \langle e \rangle \sim \langle e \rangle \mid \\
& \llbracket e \rrbracket \triangleright \text{TensorType}(\zeta_1, \zeta_2, \zeta_3, \zeta_4) \mid \\
& \llbracket e \rrbracket = \langle e \rangle \sqcup^* \langle e \rangle \mid \text{can-reshape}(\llbracket e \rrbracket, \delta) \mid \\
& \llbracket e \rrbracket = \text{calc-conv}(\llbracket e \rrbracket, c_{out}, \kappa) \mid \\
& \langle e \rangle, \langle e \rangle = \text{apply-broadcasting}(\llbracket e \rrbracket, \llbracket e \rrbracket)
\end{aligned}$$

$$\begin{aligned}
(\text{Target Constraints}) \quad \psi ::= & \quad \psi \wedge \psi \mid \psi \vee \psi \mid \neg \psi \mid \text{True} \mid \\
& v = \text{TensorType}(\zeta, \dots, \zeta) \mid \\
& v = \text{Dyn} \mid v = v \mid \zeta = n \mid \zeta = \text{Dyn} \mid \zeta = \zeta \mid \\
& \zeta = \zeta \cdot n + n \mid (\zeta_1 \cdot \dots \cdot \zeta_m) \text{ mod } (\zeta'_1 \cdot \dots \cdot \zeta'_n) = 0
\end{aligned}$$

 **Figure 4** Source constraints and target constraints.

that we have a constructor for each of our ranks, of the form `TensorType1`, `TensorType2`, `TensorType3`, and `TensorType4`. Each of the functions take a list of dimensions. Moving on to the dimensions, we have that dimensions are either `Dyn` or natural numbers. We can easily represent natural numbers in an SMT solver but we must also represent `Dyn`. One way to encode a `Dyn` dimension d is as a pair (d_1, d_2) . If $d_1 = 0$, then $d = \text{Dyn}$. Otherwise, d is a number, and its value is in d_2 . Let us formalize the constraint generation process next.

From p , we generate constraints $\text{Gen}(p)$ as follows. Let p have the form `decl* return e`. Let X be the set of declaration-variables x occurring in e , and let Y be a set of variables disjoint from X consisting of a variable $\llbracket e' \rrbracket$ for every occurrence of the subterm e' in e . Let Z be a set of variables disjoint from X and Y consisting of a variable $\langle e_1 \rangle, \langle e_2 \rangle$ for every occurrence of the subterm `add`(e_1, e_2) in e . Finally, let V be a set of variables disjoint from X , Y , and Z consisting of dimension variables ζ . The notations $\llbracket e \rrbracket$ and $\langle e \rangle$ are ambiguous because there may be more than one occurrence of some subterm e' in e or some subterm e'' in e . However, it will always be clear from context which occurrence is meant. For every occurrence of ζ , it is implicit that we have a constraint $0 \leq \zeta$ to ensure that the solver assigns a dimension in \mathbb{N} . We omit writing this explicitly for simplicity. With that in mind, we generate the constraints in Figure 5. Let us go over the rules in Figure 5. The rules use judgments of the form $\vdash x : \tau : \psi$ for declarations, and it uses judgments of the form $\vdash e : \psi$ for expressions. In both cases, ψ is the generated constraint.

$t\text{-}decl$ uses the precision relation \sqsubseteq to insure that a migration will have a more precise type, while $t\text{-}var$ propagates the type information from declarations to the program.

$t\text{-}reshape$ considers all possibilities of reshaping any tensor e with rank, at most 4, via the constraint $\llbracket e \rrbracket \leq 4$. This restriction constraint captures all rank possibilities for $\llbracket e \rrbracket$ in addition to $\llbracket e \rrbracket$ being `Dyn`. For each possibility, the number of occurrences of `Dyn` in δ and $\llbracket e \rrbracket$ varies. This impacts the arithmetic constraints that make reshaping possible, as we can see from the typing rules. As such, `can-reshape` simulates all such possibilities and generates the appropriate constraints.

$t\text{-}conv$ contains matching and consistency constraints, to model matching and consistency in convolution's typing rule. We have a constraint `calc-conv`, which generates the appropriate arithmetic constraints for the output of the convolution, based on the convolution typing rule, again accounting for the possibility of the input e having a gradual type.

$$\begin{array}{c}
\frac{}{\vdash x : \tau : \tau \sqsubseteq x \wedge |x| \leq 4} \text{ (t-decl)} \quad \frac{}{\vdash x : x = \llbracket x \rrbracket} \text{ (t-var)} \\
\\
\frac{\vdash e : \psi}{\vdash \text{reshape}(e, \delta) : \psi \wedge \llbracket \text{reshape}(e, \delta) \rrbracket = \delta \wedge \text{can-reshape}(\llbracket e \rrbracket, \delta) \wedge |\llbracket e \rrbracket| \leq 4} \text{ (t-reshape)} \\
\\
\frac{\vdash e : \psi}{\vdash \text{Conv2D}(c_{in}, c_{out}, \kappa, e) : \psi \wedge \llbracket e \rrbracket \triangleright \text{TensorType}(\zeta_1, \zeta_2, \zeta_3, \zeta_4) \wedge c_{in} \sim \zeta_2 \wedge \llbracket \text{Conv2D}(c_{in}, c_{out}, \kappa, e) \rrbracket = \text{calc-conv}(\llbracket e \rrbracket, c_{out}, \kappa)} \text{ (t-conv)} \\
\\
\frac{\vdash e_1 : \psi_1 \quad \vdash e_2 : \psi_2}{\vdash \text{add}(e_1, e_2) : \psi_1 \wedge \psi_2 \wedge \llbracket \text{add}(e_1, e_2) \rrbracket = \langle e_1 \rangle \sqcup^* \langle e_2 \rangle \wedge (\langle e_1 \rangle, \langle e_2 \rangle) = \text{apply-broadcasting}(\llbracket e_1 \rrbracket, \llbracket e_2 \rrbracket) \wedge \langle e_1 \rangle \sim \langle e_2 \rangle \wedge |\llbracket e_1 \rrbracket| \leq 4 \wedge |\llbracket e_2 \rrbracket| \leq 4 \wedge |\llbracket \text{add}(e_1, e_2) \rrbracket| \leq 4} \text{ (t-add)}
\end{array}$$

■ **Figure 5** Constraint generation.

t-add contains least upper bound constraints and consistency constraints, similar to the *add* typing rule. We constrain the inputs e_1 and e_2 , as well as the expression itself, $\text{add}(e_1, e_2)$ to all be either *Dyn* or tensor of at most rank-4, via a \leq constraint. We use the function *apply-broadcasting*, which simulates broadcasting on the shapes, on dummy variables $\langle e_1 \rangle$ and $\langle e_2 \rangle$ (notice that the real shapes of e_1 and e_2 are represented by $\llbracket e_1 \rrbracket$ and $\llbracket e_2 \rrbracket$). We check $\langle e_1 \rangle$ and $\langle e_2 \rangle$ for consistency and obtain the least upper bound.

Let φ be a mapping from tensor-type variables to $\text{TensorType}(\text{list}(\zeta)) \cup \{\text{Dyn}\}$, and also from dimension-type variables to $\text{IntConst} \cup \{\text{Dyn}\}$. We define that a target constraint ψ has solution φ , written $\varphi \models \psi$, in the following way:

The following is true:	Provided:
$\varphi \models \psi \wedge \psi'$	$\varphi \models \psi \text{ and } \varphi \models \psi'$
$\varphi \models \psi \vee \psi'$	$\varphi \models \psi \text{ or } \varphi \models \psi'$
$\varphi \models \neg \psi$	$\text{not } (\varphi \models \psi)$
$\varphi \models \text{True}$	always
$\varphi \models v = \text{TensorType}(\zeta_1, \dots, \zeta_n)$	$\varphi(v) = \text{TensorType}(\varphi(\zeta_1), \dots, \varphi(\zeta_n))$
$\varphi \models v = \text{Dyn}$	$\varphi(v) = \text{Dyn}$
$\varphi \models v = v'$	$\varphi(v) = \varphi(v')$
$\varphi \models \zeta = n$	$\varphi(\zeta) = n$
$\varphi \models \zeta = \text{Dyn}$	$\varphi(\zeta) = \text{Dyn}$
$\varphi \models \zeta = \zeta'$	$\varphi(\zeta) = \varphi(\zeta')$
$\varphi \models \zeta = \zeta \cdot n + n'$	$\varphi(\zeta) = \varphi(\zeta') \cdot n + n'$
$\varphi \models (\zeta_1 \cdot \dots \cdot \zeta_m) \text{ mod } (\zeta'_1 \cdot \dots \cdot \zeta'_n) = 0$	$(\varphi(\zeta_1) \cdot \dots \cdot \varphi(\zeta_m)) \text{ mod } (\varphi(\zeta'_1) \cdot \dots \cdot \varphi(\zeta'_n)) = 0$

► **Definition 4.** $\varphi \leq \varphi'$ iff $\text{dom}(\varphi) = \text{dom}(\varphi') \wedge \forall x \in \text{dom}(\varphi) : \varphi(x) \sqsubseteq \varphi'(x)$

Let $\text{Gen}(P)$ be the constraint generation function and $\text{Sol}(C)$ be the set of solutions to constraints C . Then we can state the order-isomorphism theorem as follows:

► **Theorem 5** (Order-Isomorphism).

$\forall P : (\text{Mig}(P), \sqsubseteq)$ and $(\text{Sol}(\text{Gen}(P)), \leq)$ are order-isomorphic.

The order-isomorphism theorem states that we have captured the migration-space with our constraints such that, for a given program, the solution space and the migration-space are order-isomorphic. For the proof, see Appendix F.

Our algorithm for code annotation is shown in Algorithm 1.

Algorithm 1 Code annotation.

Input: Program P

Output: Annotated program P'

- 1: **Constraint Generation.** Generate constraints $C = \text{Gen}(P)$.
 - 2: **Constraint Solving.** Solve C and get a solution φ that maps variables to types.
 - 3: **Program Annotation.** In P , replace each declaration $x : \tau$ with $x : \varphi(x)$, to get P' .
-

Let us now revisit Listing 1 but this time with variable x annotated by `Dyn`. We will show how to migrate a calculus version of the program by generating constraints and passing them to an SMT solver. Let us recall that this listing had two expressions that map to the following expressions in our calculus: `Conv2D(2, 2, (2, 2), x)` and `Conv2D(4, 2, (2, 2), x)`.

The first step is to generate high-level constraints:

$$\text{Dyn} \sqsubseteq v_1 \quad (1)$$

$$v_1 \leq 4 \quad (2)$$

$$v_1 \triangleright \text{TensorType}(\zeta_3, \zeta_4, \zeta_5, \zeta_6) \quad (3)$$

$$2 \sim \zeta_4 \quad (4)$$

$$v_2 = \text{calc-conv}(v_1, 2, (2, 2), (2, 2), (2, 2), (2, 2)) \quad (5)$$

$$v_1 \triangleright \text{TensorType}(\zeta_9, \zeta_{10}, \zeta_{11}, \zeta_{12}) \quad (6)$$

$$4 \sim \zeta_{10} \quad (7)$$

$$v_8 = \text{calc-conv}(v_1, 2, (2, 2), (2, 2), (2, 2), (2, 2)) \quad (8)$$

Let us go over what each equation is for. Constraint (1) denotes that the type annotation for the variable x must be as precise or more precise than `Dyn`. Constraint (2) denotes that the type annotation for x could either be `Dyn` or a tensor with at most four dimensions. We use the \leq notation to denote this. Notice that the type variable for x is v_1 . Constraints (3), (4), and (5) are for `Conv2D(2, 2, (2, 2), x)`, while constraints (6), (7), and (8) are for `Conv2D(4, 2, (2, 2), x)`. More specifically, constraints (3) and (6) determine the input shape of a convolution while constraints (5) and (8) determine the output shape of a convolution.

The main differences between the constraints for our core calculus and the ones in our implementation is that `calc-conv` takes some additional parameters in our implementation because we have implemented the full version of convolution.

The constraints above are high-level constraints which are yet to be expanded. For example, \triangleright and \leq constraints get transformed to equality constraints. We will skip writing out the resulting constraints for simplicity. After expanding these constraints and running them through an SMT solver, we get a satisfying assignment. In case multiple satisfying assignments exist, we use the one that the SMT solver picks. The fact that we got a satisfying assignment lets us know that the migration space is non-empty, which means that the program is well-typed. Let us go through some of relevant assignments:

$$\begin{aligned} \varphi(v_1) &= \text{Dyn} \\ \varphi(v_2) &= \text{TensorType}(\text{Dyn}, 2, \text{Dyn}, \text{Dyn}) \\ \varphi(v_8) &= \text{TensorType}(\text{Dyn}, 2, \text{Dyn}, \text{Dyn}) \end{aligned}$$

Here, v_1 is the type of x , v_2 is the type of the first convolution and v_8 is the type of the second convolution. We can see that these assignments are a valid typing to the program because the outputs of both convolutions should be 4-dimensional tensors with the second dimension being 2, which stands for the output channel. And since the input x has been assigned `Dyn` by our SMT solver, we cannot determine the last two dimensions of a convolution output. While this is a reasonable output, it may not be helpful to the programmer. Furthermore, this program would not accept any concrete output. We know this from our constraints. From constraints (3) and (7), we have that $\zeta_4 = \zeta_{10}$. Then from (4), (8), which are $2 \sim \zeta_4$ and $4 \sim \zeta_{10}$, we can see that the only satisfying solution is `Dyn`. This means that the program cannot be statically typed. Next, we will see how to prove this formally.

Let us discuss how to extend our approach to solve Q(1) and Q(2). In the example above, the migration space is non-empty and we may want to know if we can statically type the program. We have established that we cannot. As a first step, we may want to take our core constraints above, which we will call C , and restrict the input to a rank-4 tensor. So we can consider the constraint $C \wedge x = \text{TensorType}(\zeta'_1, \zeta'_2, \zeta'_3, \zeta'_4)$ where $\zeta'_1, \dots, \zeta'_4$ are fresh variables. We can begin to impose restrictions on $\zeta'_1, \dots, \zeta'_4$ to make them concrete variables. For example, if we restrict the last dimension to be a number, we can add the constraint $\zeta'_4 \neq \text{Dyn}$. After running our constraints through the solver, we get the following assignments:

$$\begin{aligned}\varphi(v_1) &= \text{TensorType}(\text{Dyn}, \text{Dyn}, \text{Dyn}, 28470) \\ \varphi(v_2) &= \text{TensorType}(\text{Dyn}, 2, \text{Dyn}, 14236) \\ \varphi(v_8) &= \text{TensorType}(\text{Dyn}, 2, \text{Dyn}, 14236)\end{aligned}$$

To prove that no concrete assignment to the second dimension of x is possible, we simply add $\zeta'_2 \neq \text{Dyn}$ to our original constraints and the constraints will be unsatisfiable, so we conclude that the second dimension of x can only be `Dyn`.

We can also answer Q(2) by feeding the solver additional arithmetic constraints about dimensions. In our example, if we want the first dimension of x to be between 3 and 10, we can add the constraint $\zeta'_1 \leq 3 \wedge \zeta'_1 \geq 10$ to $C \wedge x = \text{TensorType}(\zeta'_1, \zeta'_2, \zeta'_3, \zeta'_4)$ and rerun our solver.

Our migration solution is based on a satisfiability problem: *is our migration problem decidable?* If so, what is the time complexity? The migration problem is decidable if the underlying constraints are drawn from a decidable theory. Those underlying constraints are the ones given by the grammar in Section 4. Let us for a moment ignore constraints of the form $(\zeta_1 \cdot \dots \cdot \zeta_m) \text{ mod } (\zeta'_1 \cdot \dots \cdot \zeta'_n) = 0$. We observe that all the other constraints are drawn from a well-known decidable theory. Specifically, the other constraints are drawn from quantifier-free Presburger arithmetic extended with uninterpreted functions and equality. The satisfiability problem for this theory is NP-complete [21]. Once we add constraints of the form $(\zeta_1 \cdot \dots \cdot \zeta_m) \text{ mod } (\zeta'_1 \cdot \dots \cdot \zeta'_n) = 0$, the decidability-status of the satisfiability problem is unknown, to the best of our knowledge. Fortunately, only three operations need this additional constraint: `Reshape`, `View`, or `Flatten`. All the other 47 operations that our implementation supports need only constraints in the NP-complete subset. Our implementation translates all of the constraints to Z3 format, and while our benchmarks do need constraints outside the NP-complete subset, our experiments terminated. In every case, Z3 terminated with either sat or unsat. Thus, the generated constraints are simple enough for Z3 to solve, even if the general case is undecidable.

The complexity of migration depends on the size of the constraint we generate. The bottleneck is the \leq constraint; let us see how to expand it.

From: $\|\llbracket e \rrbracket\| \leq 4$

To: $\llbracket e \rrbracket = \text{Dyn} \vee \llbracket e \rrbracket = \text{TensorType}(\zeta_1) \vee \dots \vee \llbracket e \rrbracket = \text{TensorType}(\zeta_1, \zeta_2, \zeta_3, \zeta_4)$

where ζ_1, \dots, ζ_4 are fresh variables. This yields a complexity of 4^n in the number of \leq constraints. So assuming that any additional constraints are drawn from the NP-complete subset, the problem will still be decidable. Note that if we are working with a fixed rank, then these constraints will be generated in polynomial time in the size of the program. Below we will see how solving the problem for a fixed rank has practical benefits.

5 Extending our approach to do Branch Elimination

We introduce our approach to branch elimination via the following example.

```

1  class ReshapeControlFlow(torch.nn.Module):
2      def __init__(self):
3          super().__init__()
4
5      def forward(self, x: Dyn):
6          if x.reshape(100).size()[0] < 100:
7              return torch.dropout(x, p=0.5, train=False)
8          else:
9              return torch.relu(x)

```

Listing 7 An example of graph-break elimination

In contrast to listing 5, where the conditional depends of the rank of the input, listing 7 has a conditional that depends on the value of one of the dimensions in the input shape. Listing 7 uses the `reshape` function, which takes a tensor and re-arranges its elements according to the desired shape. In this case, we reshape `x` to have the shape `TensorType([100])`. For reshaping to succeed, the initial tensor must contain the same number of elements as the reshaped tensor. Notice that since `x` is typed as `Dyn`, the program will type check. In the expression `x.reshape(100).size()`, the expression `size()` will return the shape of `x.reshape(100)`, which is `[100]`. We are then getting the first dimension of the shape in the expression `x.reshape(100).size()[0]`, which is 100. Thus, by inspecting the conditional `if x.reshape(100).size()[0] < 100`, we can see that the conditional should always evaluate to false. Thus, we can remove the true branch from the program and produce listing 8. In contrast, TorchDynamo breaks Listing 7 into two different programs: one for when the condition evaluates to true, and another for when the condition evaluates to false.

```

1  class ReshapeControlFlow(torch.nn.Module):
2      def __init__(self):
3          super().__init__()
4
5      def forward(self, x: Dyn):
6          return torch.relu(x)

```

Listing 8 An example of graph-break elimination

Let us see an example of how to extend our constraint-based solution to eliminate the extra branch. For listing 7, here are the constraints for `x.reshape(100).size()[0]` in line 6. The variable ζ_4 is for the result of the entire expression. Note that the PyTorch expression `x.reshape(100)` is the same as the calculus expression `reshape(x, TensorType(100))`.

$$\text{Dyn} \sqsubseteq v_1 \wedge v_1 \leq 4 \quad (1)$$

$$v_2 = \text{TensorType}(100) \wedge \text{can-reshape}(v_1, \text{TensorType}(100)) \quad (2)$$

$$v_2 = v_3 \quad (3)$$

$$(v_3 = \text{Dyn} \wedge \zeta_4 = \text{Dyn}) \vee ((\zeta_4 = \text{GetItem}(v_3, 1, 0) \vee \zeta_4 = \text{GetItem}(v_3, 2, 0) \vee \\ \zeta_4 = \text{GetItem}(v_3, 3, 0) \vee \zeta_4 = \text{GetItem}(v_3, 4, 0)) \quad (4)$$

Above, the constraint (1) is for x . Notice that v_1 is the type variable for x . Constraint (2) is for `reshape(x, TensorType(100))`. Next, when encountering the `size` function in a program, we simply propagate the shape at hand with an equality constraint, which is seen in equation (3). If we are indexing into a shape, we consider all the possibilities for the sizes of that shape and generate constraints accordingly. In particular, we have that ($v_3 = \text{Dyn} \wedge \zeta_4 = \text{Dyn}$) because a shape could be dynamic, which means that if we index into it, we get a `Dyn` dimension. But since we restricted our rank to 4, we can consider the possibilities of the index being 1, 2, 3 or 4, which is what the remaining constraints do.

We extend our constraint grammar with constructs that enable us to represent `size()` and indexing into shapes. This includes constraints of the form $\zeta = \text{GetItem}(v, c, i)$, where v is the shape we are indexing into, c is the assumed tensor rank, and i is the index of the element we want to get. We can map the new constraints to Z3 constraints easily.

Next we generate a constraint ($\zeta_4 < 100$) for the condition and a constraint $\neg(\zeta_4 < 100)$ for its negation. If C are the constraints for the program up to the point of encountering a branch, then we generate both $C \wedge \zeta_4 < 100$ and $C \wedge \neg(\zeta_4 < 100)$.

We evaluate both sets of constraints. One set must be satisfiable while the other must be unsatisfiable for us to remove the branch. If we are unable to remove the branch, this means that the input set is still too general such that for some inputs, the branch may evaluate to true and for other inputs, the branch may evaluate to false. In such case, we can ask the user to capture a stricter subset of the input by further constraining it. We can then re-evaluate our constraints again to see if we are able to remove the branch.

We extend our grammar with conditional expressions `if cond then e1 else e2`. Algorithm 2 describes how to eliminate a single branch.

Algorithm 2 Branch elimination.

Input: Program p .

Output: A possibly modified p with a branch eliminated.

- 1: Let C = the constraints for p up to encountering a branch `if cond then e1 else e2`.
 - 2: Let c_{cond} = the constraints for `cond`.
 - 3: **if** $(C \wedge c_{cond})$ is satisfiable and $(C \wedge \neg c_{cond})$ is unsatisfiable **then**
 - 4: Rewrite the branch to e_1
 - 5: **else if** $(C \wedge c_{cond})$ is unsatisfiable and $(C \wedge \neg c_{cond})$ is satisfiable **then**
 - 6: Rewrite the branch to e_2
 - 7: **else**
 - 8: Require the user to change the shape information
 - 9: **end if**
-

6 Implementation

PyTorch has three tool-kits that rely on symbolic tracers [3]. Let us go over each one. First, `torch.fx` [17] is a common PyTorch tool-kit and has a symbolic tracer. Symbolic tracing is a process of extracting a more specialized program representation from a program, for the purpose of analysis, optimization, serialization, etc. `torch.fx` does not accept programs containing branches and the `torch.fx` authors emphasize that “*most neural networks are expressible as flat sequences of tensor operations without control flow such as if-statements or loops*” [17]”. HFtracer [29] eliminates branches by symbolically executing on a single input. Finally, TorchDynamo [2] handles dynamic shapes by dividing the program into fragments.

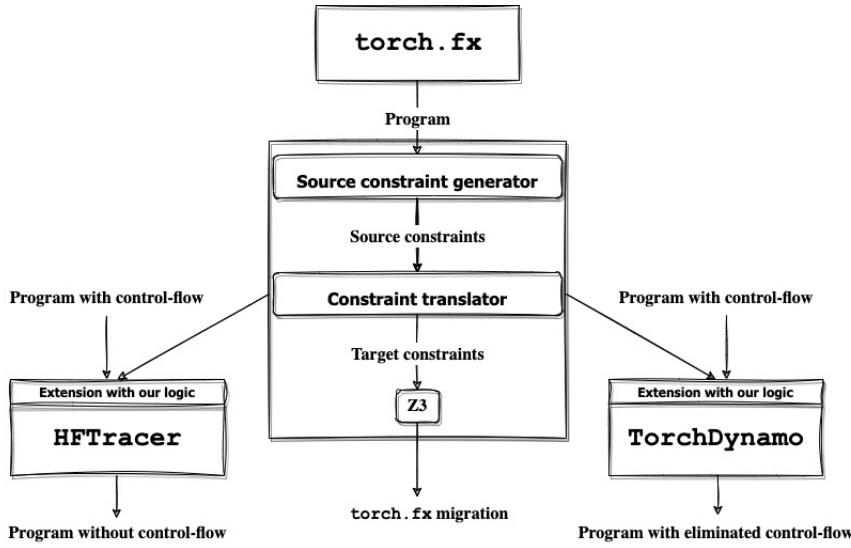


Figure 6 Our core tool and the three tracers.

This process is called a *graph-break*. Specifically, when encountering a condition that depends on shape information and where shape information is unknown, the program is broken into two parts. One fragment is for when the result of the condition is true, and another is for when the result of the condition is false. Graph-breaks result in multiple programs with no branches.

As a technical detail, code annotation for the purpose of program understanding and better documentation is meant to be performed on a source language; branch elimination is done at trace-time, on an intermediate representation. For the purpose of better readability, we presented all the examples in Section 2 in source code syntax. In some of our larger benchmarks, the source code is different from the intermediate representation because more high-level constructs were used, such as statements. However, statements do not influence our theoretical results. We did not include sequences in our theory because they did not introduce additional challenges to our problem. Finally, there are some constructs in PyTorch that propagate variable shapes, such as `dim()` and `size()`. There are also getters which index into shapes. Those constructs were used to write ad-hoc shape-checks. We dealt with them in our implementation by propagating shape information accordingly.

We have implemented approximately 6000 LOC across three different tracers. Figure 6 summarizes how our implementation works. First, we implement a core constraint generator. This generator takes a program (in our benchmarks case, a program is generated via `torch.fx`), and generates core, source constraints for it. Next is the constraint translator which consists of two phases. In the first phase, it encodes the gradual types found in the program then translates the source constraints into target constraints. Note that a program is annotated, possibly with a `Dyn` type for every variable. In the second phase, it translates the target constraints into `Z3` constraints, which is a 1:1 translation.

Next, we modify each of `TorchDynamo` and `HFtracer` to incorporate our reasoning and use it for branch elimination. We must incorporate our logic into the tracers because *branch elimination happens at trace-time*, unlike program migration which requires a whole program.

Our implementation faithfully follows our core logic, although we have made some practical simplifications. First, our implementation focuses on supporting 50 PyTorch operations that our benchmarks use. Each of those operations has its own constraints and supporting all 50

was multiple months of effort. Second, for the `view` operation (which is similar to `reshape` in terms of types, see <https://pytorch.org/docs/stable/generated/torch.Tensor.view.html>), we have skipped implementing dynamism and required the solver to provide concrete dimensions. This allowed us to carry out branch elimination without requiring an additional constraint that disables dynamism, although the same effect can be accomplished in this manner as well. Third, `Conv2D` may accept rank-3 or rank-4 inputs, but we have limited our implementation to the rank-4 case, since this is the case that is relevant to most of our benchmarks.

We ran our experiments on a MacBook Pro with an 8-Core CPU, 14-Core GPU and 512GB DRAM.

7 Experimental Results

We answer the following three questions.

- Q(1): Can our tool determine if the migration space is non-empty? If so, can it determine if the migration space contains a static migration and if so, can it find one? *Yes. Our tool is the first to affirmatively answer all three questions.*
- Q(2): Given an arithmetic constraint on a dimension, can our tool determine if there is a migration that satisfies it and if so, can it find one? *Yes. Our tool is the first to retrieve migrations that provably satisfy arbitrary arithmetic constraints.*
- Q(3): Can our tool prove that branch elimination is valid for an infinite set of inputs, not just for a single input? If so, does it allow us to represent the set of inputs for which a branch evaluates to true or false? *Yes. We incorporate our logic into two different tools and eliminate branches in all benchmarks we considered for infinite classes of input, characterized via constraints. Neither tool was able to achieve this without our logic.*

Figure 7 contains our benchmark names, the source of the benchmark, lines of code, and the number of `flatten` and `reshape` operations in each benchmark. The `flatten` and `reshape` operations are special because our analysis of them involves multiplication and modulo constraints. Our benchmarks are drawn from two well-known libraries, TorchVision and Transformers [30, 29], with the exception of two microbenchmarks that we use as examples in Section 2. We used different benchmarks for different experiments. The first four models do not contain branches, making them suitable for Q(1) and Q(2). They are interesting because `BmmExample` has a shape mismatch, `ConvExample` cannot be statically migrated, and `AlexNet` and `ResNet50` are well-known neural-network models. Our experience is that tensor programs are tricky to type, and that our tool offers feedback that helps the user narrow down the migration space by adding constraints. The next six models are suitable for our `HFTtracer` experiments. Those experiments required reasoning about whole programs and our tool was able to reason about them in under two minutes. The final four benchmarks are of a larger size. We do not support all the operations in those benchmarks. However, this did not pose a problem because in `TorchDynamo`, we were not required to reason about entire programs. Instead, we were required to reason about program fragments, which made our tool terminate in under three minutes.

We ran our tool in the following way to answer Q(1).

1. Generate the core constraints and check if they are satisfiable. If not, stop right away; The program is ill-typed.
2. Determine if the input variable can have a concrete rank by asking the solver for migrations of concrete ranks from one to four. If none exist, the input variable was used at different ranks throughout the program.

29:22 Generalizing Shape Analysis with Gradual Types

Benchmark	Source	LOC	Flatten	Reshape	Used for
BmmExample	this paper	4	0	0	Q(1)
ConvExample	this paper	6	0	0	Q(1)
AlexNet	TorchVision	24	1	0	Q(1)
ResNet50	TorchVision	177	1	0	Q(1)
Electra	Transformers	525	0	48	Q(2)
Roberta	Transformers	533	0	48	Q(2)
MobileBert	Transformers	2103	0	96	Q(2)
Bert	Transformers	528	0	48	Q(2)
MegatronBert	Transformers	1018	0	96	Q(2)
XGLM	Transformers	104	0	14	Q(2) and Q(3)
Marian	Transformers	1733	0	315	Q(3)
MarianMT	Transformers	1735	0	315	Q(3)
M2M100	Transformers	1762	0	319	Q(3)
BlenderBot	Transformers	2380	0	451	Q(3)

Figure 7 Benchmark information.

Benchmark	Q(1)		Q(2)	
	Static migration?	Time(s)	Arithmetic constraints?	Time(s)
BmmExample	No	0.03	No	0.03
ConvExample	No	0.05	Yes	0.08
AlexNet	Yes	2	Yes	2
ResNet50	Yes	5	Yes	347

Figure 8 Q(1) and Q(2): static migration and migration under arithmetic constraints.

3. If the input variable can be assigned concrete ranks, pick one of them and ask the tool to statically annotate all dimensions.
4. If the solver cannot statically annotate all dimensions, relax this requirement for each dimension to determine which one cannot be statically annotated.

We first traced our benchmarks using `torch.fx`, then ran the above steps on the output. The first step simply involves running our tool, while the second and third steps require the user to pass constraints to the tool and rerun it. Determining if a variable has a certain rank requires a single run with our tool. Determining if a dimension can be static requires a single run with our tool. The final step involves removing constraints. Each time we remove a constraint from a dimension, we can run our tool once to determine a result.

The first part of Figure 8 summarizes our results. The first column in the figure is the benchmark name. The second column asks if the benchmark has a static migration and the third column measures the time it took to answer this question and retrieve a static migration. For ConvExample, the input can only be rank-4 and the second dimension can only be `Dyn`. BmmExample has a type error. Finally, ResNet50 and AlexNet can be fully typed and the inputs can only be rank-4 in both cases.

We ran our tool in the following way to answer Q(2). First we follow the steps for answering Q(1), and if any dimensions can be static, then we apply further arithmetic constraints on some of those dimensions and ask for a migration that satisfies them. We ran the steps above in our extension of `torch.fx`. The second part of Figure 8 summarizes our results. The fourth column asks if arithmetic constraints can be imposed on at least one

of the dimensions and the fifth column measures the time it took to answer this question and retrieve a migration that satisfies an arithmetic constraint. For ResNet50 and AlexNet, we added arithmetic constraints. For ConvExample, we fixed the example like we did in Section 2 then added arithmetic constraints. We obtained valid migrations that satisfy our constraints for all benchmarks, except for BmmExample which is ill-typed and thus has an empty migration space.

We ran our tool in the following way to answer Q(3). We ran our extension of **HFtracer**, starting with annotating the input with **Dyn** and then gradually increasing the precision of our constraints to provide the solver with more information to eliminate more branches. The number of times we run our tool here depends on how much information the user gives the tool about the input. If the tool receives static input dimensions, then this will be enough to eliminate all branches that depend on shapes. But since we aim to relax this requirement, we could start with a **Dyn** shape then gradually impose constraints, first with rank information, then with dimension information.

We were able to eliminate all branches this way. We followed similar steps in our **TorchDynamo** extension but we faced some practical concerns because **TorchDynamo** currently does not carry parameter information between program fragments. We had to resolve this issue manually by passing additional constraints at every new program fragment.

Figure 9 details our **HFtracer** experiments on 6 workloads. Figure 9 contains the original number of branches in the program, the remaining branches after running our extension, without imposing any constraints on the input, and the number of remaining branches after running our extension, with the constraints in Figure 9 on the input. The second-to-last column of the figure is the time it takes to perform branch elimination with constraints.

HFtracer also eliminates all branches from the 6 workloads. However, it does this by running the program on an input. We can obtain a similar result by giving a constraint describing the *shape* of the input because we observed that for all benchmarks we considered, an actual input is not needed to eliminate all branches, and we can relax this requirement much further. Specifically, for some benchmarks, no constraints are needed at all to eliminate all branches, while for others, it is enough to specify rank information. For one of the benchmarks, we can specify a range of dimensions for which branches can be eliminated. Figure 9 details the constraints.

Finally figure 10 represents branch elimination for **TorchDynamo**. There are two modes of operation in **TorchDynamo** called static and dynamic. In the static mode, the tracer traces the program with one input which is provided by the user. Branch elimination is therefore valid for a single input. In Dynamic mode, the tracer also takes an input but it only records *rank* information and ignores the values of the dimensions. So if a branch depends on dimension information, a graph-break will occur. We focused on benchmarks where branches depend on dimension information. In figure 10, we impose constraints on the dimensions and eliminate branches which decreases the number of times **TorchDynamo** breaks the program when tracing. The first column in the figure indicates the benchmark names. Next is the original number of branches with **TorchDynamo**. Then we have the remaining number of branches after incorporating our reasoning. Finally, we measure time in seconds. The input constraints are range and rank constraints, as exemplified by the constraints for XGLM shown in Figure 9.

From our experiments, we observed that slowdowns can be due to the kind of constraints involved and the number of constraints to solve. Our tool typically handles benchmarks that are under 1000 lines of code easily. However, range constraints impose overhead. For example, ResNet50 and XGLM contain such constraints and they were the slowest in Figure

Benchmark	# remaining branches			Time (s)	our constraints
	without original	with constr.	with constr.		
Electra	3	3	0	1	$\text{Tensor}(x, y)$
Roberta	3	0	0	3	none
MobileBert	3	3	0	1	$\text{Tensor}(x, y)$
Bert	3	0	0	3	none
MegatronBert	3	0	0	5	none
XGLM	5	4	0	22	$\text{Tensor}(x, y) \wedge x > 0 \wedge 1 < y < 2000$

Figure 9 Q(3): HFtracer number of remaining branches.

Benchmark	original	with constraints	Time(s)
XGLM	5	0	45
Marian	44	26	70
MarianMT	44	26	75
M2M100	47	22	130
BlenderBot	35	19	40

Figure 10 Q(3): TorchDynamo number of remaining branches.

9. For the experiments under Q(1) and Q(2), we let the tools run more than 5 minutes, but for Q(3) we limit to 5 minutes. The benchmarks in figure 10 are over 1000 lines, and for some branches, branch elimination with TorchDynamo times out after 5 minutes.

There are two limitations to our TorchDynamo experiments. First, since PyTorch has various operations with many layers of abstractions and edge cases, not every edge case was implemented. Given that this only affected a few branches, we chose to skip those branches. This did not affect our experiments because TorchDynamo does not require all branches to be removed. Each branch removed will result in one less graph-break. TorchDynamo induces graph-breaks for reasons other than control flow. When graph-breaks happen, we have to re-write an input constraint for the resulting fragments because there is currently no clear mechanism in passing parameter information from one fragment to another. We manually passed input constraints to program fragments until eliminating at least 40% of branches and have stopped after that due to the large size of the benchmarks and program fragments. We leave parameter preservation during graph-breaks to the TorchDynamo developers.

8 Related work

We first discuss related work about shapes in tensor programs.

[15] show how to do shape checking based on assertions written by programmers. Their assertions can reason about tensor ranks and dimensions, with arithmetic constraints. Our work also supports such constraints. Their tool executes a program symbolically and looks for assertion violations. The more assertions programmers write, the more shape errors their tool can report. Their tool uses Z3 to solve constraints of a size that can be up to exponential in the size of the program. Our approach is similar in that it enables programmers to annotate a program with types and to type check the program and thereby catch shape errors. Another similarity is that we use Z3 to solve constraints of exponential size. Our approach differs by going further: we have tool support for annotating any program with types and for removing unnecessary runtime shape checks. Additionally, we have proved that our type system has key correctness properties.

[9] define a gradually typed system for tensor computations and, like us, they prove that it has key correctness properties. They use refinement types to represent tensor shapes, they enable programmers to write type annotations, and they do best-effort shape inference. Their refinements share some characteristics with the assertions used by [15], as well as with our constraints. They found that, for each of their benchmarks, few annotations are sufficient to statically type check the entire program. They focus on shape checking and shape inference, while we focus on generalizing shape analysis for various tasks including program migration and branch elimination. Their approach adds the traditional gradual runtime checks [22] in cases where annotations and shape inference fall short. Our work differs by enabling program optimizations through removing runtime checks, while we leave out gradual runtime checks. Conceptually, our approach and the one from [9] differ in that we define type migration syntactically, while they follow a semantic interpretation of gradual types. It is unclear how migration would be defined in their context. Another difference is that we have demonstrated scalability: their benchmark programs are up to 258 lines of code, while our benchmark programs are up to 2,380 lines of code. We were unable to do an experimental comparison because our tool works with PyTorch, while their tool works with OCaml-Torch.

[31] analyzed the root causes of bugs in TensorFlow programs by scanning StackOverflow and GitHub. They identified four symptoms and seven root causes for such bugs. The most common symptoms are functional errors, crashes, and build failure, while common root causes are data processing errors, type confusion, and dimension mismatches. Our type system can help spot those root causes because key parts of such code will have type `Dyn`, even after migration.

[11] use static analysis to detect shape errors in TensorFlow. Their approach statically detects 11 of the 14 TensorFlow bugs reported by [31], but has no proof of correctness. Our approach differs from [11] by being able to annotate a program with types and being able to remove unnecessary runtime checks. Our work can reason about programs without requiring any type annotations and only taking into account the shape information from the operations used in the program, while [11] requires a degree of type information. In contrast, we have proved that our type system has key migratory properties, such as that our constraints represent the entire migration space for a program, allowing us to extract and reason about all existing shape information from the program according to the programmer's needs.

[10] is a static analysis tool that detects shape errors in PyTorch programs. Their approach is different than ours in that it detects errors via symbolic execution. It considers all possible execution paths for a program to reason about shapes. The number of execution paths can be large. In contrast, our approach reasons about shapes which can be given in the form of type annotations or can be detected from the program.

[27] consider a dynamic analysis tool for TensorFlow, called ShapeFlow, to detect shape errors. The advantage of this approach is that, like our approach, it does not require type annotations, but their analysis holds for only particular inputs, in contrast to our approach, which reasons about programs across all possible inputs. Unlike our work, their approach has not been formalized, but there is empirical evidence to support that it detects shape errors in *most* cases. Because we reason about programs statically, our work is more suitable for compiler optimizations and program understanding. Our shape analysis approach can be used to annotate programs. In contrast, ShapeFlow is more suitable if a programmer desires a light-weight form for error detection that works in most cases.

[20] designed an intermediate representation called Relay. It is functional, like our calculus, but is statically-typed, unlike our gradual type system. Its goals are similar to ours in that it aims to balance expressiveness, portability, and compilation. Unlike our system, as a static

type system, Relay requires type annotations for every function parameter. Similar to our approach, their work focuses on the static aspect of the problem and has left the runtime aspect to future work.

[19] extends [20] by using a static polymorphic type system for shapes, which we leave to future work. This system has a type named `Any`, which enables partial annotations, but which appears to provide less flexibility than our `Dyn` type because of the absence of type consistency.

Next we discuss two closely related papers on migratory typing.

[12] defined the migration space for a gradually typed program as the set of all well-typed, more-precise programs. They represented the migration space for a given program by generating constraints where each solution represents a migration. The constraint-based approach enables them to solve migration problems for a λ -calculus. We adapted their definition of type migration and migration space to our context of a tensor calculus and rather different types. We use their idea of a migration space and constraints to give an algorithm that annotates a program with types and an algorithm that removes unnecessary runtime checks. In contrast to their approach, we use an SMT solver (Z3) because it can deal with the arithmetic nature of tensor constraints.

[16] build a tool which extends [12], by providing several criteria for choosing migrations from the migration space. Their work is about simple types, while our work is about tensor shapes. While their work is specifically focused on reasoning about the migration space for program annotation, we reason about the migration space more generally, by using it for general tensor reasoning tasks including program annotation and branch elimination. Their gradual language contains traditional gradual runtime checks, while we leave out runtime aspects.

9 Conclusion

We have presented a method that reasons about tensor shapes in a general way. Our method involves a gradual tensor calculus with key properties and support for decidable shape analysis for a large set of operations. Our algorithm is practical because it works on 14 non-trivial benchmarks across three different tracers. We expect that our approach to branch elimination can be extended to handle other forms of shape-based optimization.

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, Dan Mane, 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 Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems, 2016. doi:10.48550/arXiv.1603.04467.
- 2 Jason Ansel. TorchDynamo. Software release, January 2022. URL: <https://github.com/pytorch/torchdynamo>.
- 3 Jason Ansel, Animesh Jain, David Berard, Will Constable, Will Feng, Sherlock Huang, Mario Lezcano, CK Luk, Matthias Reso, Michael Suo, William Wen, Richard Zou, Edward Yang, Michael Voznesensky, Evgeni Burovski, Alban Desmaison, Jiong Gong, Kshiteej Kalambarkar, Yanbo Liang, Bert Maher, Mark Saroufim, Phil Tillet, Shunting Zhang, Ajit Mathews, Horace

- He, Bin Bao, Geeta Chauhan, Zachary DeVito, Michael Gschwind, Laurent Kirsch, Jason Liang, Yunjie Pan, Marcos Yukio Siraichi, Eikan Wang, Xu Zhao, Gregory Chanan, Natalia Gimelshein, Peter Bell, Anjali Chourdia, Elias Ellison, Brian Hirsh, Michael Lazos, Yinghai Lu, Christian Puhrsch, Helen Suk, Xiaodong Wang, Keren Zhou, Peng Wu, and Soumith Chintala. Pytorch 2: Faster machine learning through dynamic Python bytecode transformation and graph compilation. In *Proceedings of ASPLOS'24, International Conference on Architectural Support for Programming Languages and Operating Systems*, 2024.
- 4 Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, pages 257–281, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
 - 5 James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs. Software release, 2018. URL: <http://github.com/google/jax>.
 - 6 Matteo Cimini and Jeremy Siek. The gradualizer: A methodology and algorithm for generating gradual type systems. In *Proceedings of POPL'16, ACM Symposium on Principles of Programming Languages*, New York, 2016. ACM.
 - 7 Ben Greenman and Matthias Felleisen. A spectrum of type soundness and performance. *Proc. ACM Program. Lang.*, 2(ICFP), July 2018. doi:10.1145/3236766.
 - 8 Ben Greenman and Zeina Migeed. On the cost of type-tag soundness. In *PEPM*, 2018.
 - 9 Momoko Hattori, Naoki Kobayashi, and Ryosuke Sato. Gradual tensor shape checking, 2022. doi:10.48550/arXiv.2203.08402.
 - 10 Ho Young Jhoo, Sehoon Kim, Woosung Song, Kyuyeon Park, DongKwon Lee, and Kwangkeun Yi. A static analyzer for detecting tensor shape errors in deep neural network training code. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering (ICSE)*, 2022.
 - 11 Sifis Lagouvardos, Julian T Dolby, Neville Grech, Anastasios Antoniadis, and Yannis Smaragdakis. Static analysis of shape in tensorflow programs. In *ECOOP*, Germany, 2020. LIPICS.
 - 12 Zeina Migeed and Jens Palsberg. What is decidable about gradual types? *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi:10.1145/3371097.
 - 13 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 2017 Workshop on Autodiff*, 2017. URL: <https://openreview.net/forum?id=BJJsrmfCZ>.
 - 14 Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL: <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>.
 - 15 Adam Paszke and Brennan Saeta. Tensors fitting perfectly, 2021. doi:10.48550/arXiv.2102.13254.
 - 16 Luna Phipps-Costin, Carolyn Jane Anderson, Michael Greenberg, and Arjun Guha. Solver-based gradual type migration. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, New York, 2021. ACM.
 - 17 James K. Reed, Zachary DeVito, Horace He, Ansley Ussery, and Jason Ansel. Torch.fx: Practical program capture and transformation for deep learning in python. Accessed Jul 12, 2024, 2021. doi:10.48550/arXiv.2112.08429.
 - 18 Norman A. Rink. Modeling of languages for tensor manipulation. *ArXiv*, abs/1801.08771, 2018.

- 19 Jared Roesch, Steven Lyubomirsky, Marisa Kirisame, Josh Pollock, Logan Weber, Ziheng Jiang, Tianqi Chen, Thierry Moreau, and Zachary Tatlock. Relay: A high-level IR for deep learning. *CoRR*, abs/1904.08368, 2019. [arXiv:1904.08368](https://arxiv.org/abs/1904.08368).
- 20 Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. Relay: a new IR for machine learning frameworks. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. ACM, June 2018. doi:10.1145/3211346.3211348.
- 21 Sanjit A. Seshia and Randal E. Bryant. Deciding quantifier-free presburger formulas using parameterized solution bounds. *Logical Methods in Computer Science*, 1:1–26, 2005.
- 22 Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *IN SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP*, pages 81–92, 2006.
- 23 Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *SNAPL*, pages 274–293, Germany, 2015. LIPICS.
- 24 Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. Perses: Syntax-guided program reduction. In *ICSE'18, International Conference on Software Engineering*, 2018.
- 25 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*, pages 456–468, 2016.
- 26 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’08, pages 395–406, New York, NY, USA, 2008. ACM. doi:10.1145/1328438.1328486.
- 27 Sahil Verma and Zhendong Su. Shapeflow: Dynamic shape interpreter for tensorflow, 2020. doi:10.48550/arXiv.2011.13452.
- 28 Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. Big types in little runtime: Open-world soundness and collaborative blame for gradual type systems. *SIGPLAN Not.*, 52(1):762–774, 2017.
- 29 Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Perric Cistac, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45. Association for Computational Linguistics, October 2020. URL: <https://www.aclweb.org/anthology/2020.emnlp-demos.6>.
- 30 Wayne Wolf. *Computers as Components, Principles of Embedded Computing System Design*. Morgan Kaufman Publishers, 2000.
- 31 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. Association for Computing Machinery. doi:10.1145/3213846.3213866.

Verifying Lock-Free Search Structure Templates

Nisarg Patel  New York University, NY, USA

Dennis Shasha  New York University, NY, USA

Thomas Wies  New York University, NY, USA

Abstract

We present and verify template algorithms for lock-free concurrent search structures that cover a broad range of existing implementations based on lists and skip lists. Our linearizability proofs are fully mechanized in the concurrent separation logic Iris. The proofs are modular and cover the broader design space of the underlying algorithms by parameterizing the verification over aspects such as the low-level representation of nodes and the style of data structure maintenance. As a further technical contribution, we present a mechanization of a recently proposed method for reasoning about future-dependent linearization points using hindsight arguments. The mechanization builds on Iris' support for prophecy reasoning and user-defined ghost resources. We demonstrate that the method can help to reduce the proof effort compared to direct prophecy-based proofs.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Separation logic; Theory of computation → Shared memory algorithms

Keywords and phrases skip lists, lock-free, separation logic, linearizability, future-dependent linearization points, hindsight reasoning

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.30

Related Version *Extended Version:* <https://arxiv.org/abs/2405.13271> [36]

Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact):*

<https://doi.org/10.4230/DARTS.10.2.15> [35]

Software: <https://doi.org/10.5281/zenodo.11051385> [37]

Funding This work is funded in parts by NYU Wireless and by the United States National Science Foundation under grants CCF-2304758, 1840761, 2304758, and 25-74100-F1202. Further funding came from an Amazon Research Award Fall 2021. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of Amazon.

Acknowledgements We thank Sebastian Wolff for many insightful discussions and his suggestions to improve the presentation of the paper.

1 Introduction

A search structure is a key-based store that implements a mutable map of keys to values (or a mutable set of keys). It provides five basic operations: (i) create an empty structure, (ii) insert a key-value pair, (iii) search for a key and return its value, (iv) delete the entry associated with a key, and (v) update the value associated with a particular key. Because of their general usefulness, search structures are ubiquitous in data-intensive workloads.

Earlier works [19, 34, 18] developed a framework to verify a wide range of lock-based implementations of concurrent search structures. Specifically, they proved that these implementations are linearizable [11].

 © Nisarg Patel, Dennis Shasha, and Thomas Wies;
licensed under Creative Commons License CC-BY 4.0
38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 30; pp. 30:1–30:28



A core ingredient of the framework is the idea of template algorithms [39]. A template algorithm dictates how threads interact but abstracts away from the concrete layout of nodes in memory. Once the template algorithm is verified, its proof can be instantiated on a variety of search structures.

The template algorithms of [19, 34, 18] use locks as a synchronization technique. Locks ensure non-interference on portions of memory to guarantee that certain needed constraints hold in spite of concurrency.

The disadvantage of locks is that if a thread holding a lock on some portion of memory p stops, then no other thread can get a conflicting lock on p . For that reason, some practical implementations such as Java’s `ConcurrentSkipListMap` [33] use lock-free algorithms.

This paper shows how to capture multiple variants of concurrent lock-free skip lists and linked lists in the form of template algorithms. Thus, proving the correctness of such a template algorithm results in a proof that is applicable to many variants at once. Our template algorithms are parametric in the skip list height and allow variations along the following three dimensions: (i) maintenance style (eager vs lazy) (ii) node implementations and (iii) the order of maintenance operations on the higher levels of the skip lists.

By instantiating our template algorithm with appropriate maintenance operations and node implementations we obtain verified versions of existing (skip)list algorithms from the literature such as the Herlihy-Shavit skip list algorithm [10, § 14], the Michael set [31], and the Harris list algorithm [9]. We also obtain a new concurrent skip list algorithm that has not been considered before. The new algorithm is correct by construction thanks to our modular verification framework.

We mechanize our development in the concurrent separation logic Iris [14, 16]. One technical contribution of our work is a formalization of *hindsight reasoning* [32, 22, 6, 7, 26, 27] in Iris. Hindsight reasoning has shown its usefulness in dealing with future-dependent and external linearization points, a challenge that commonly arises in lock-free data structures.

Specifically, we build on the hindsight theory developed in [27], providing a mechanism in Iris where one can establish that a linearization point has passed by inferring knowledge about past states using a form of temporal interpolation.

To our knowledge, our development is the first formalization of hindsight theory in a foundational program logic. The usefulness of the developed theory extends beyond our lock-free template algorithms. In fact, we demonstrate that it can help to reduce the proof effort compared to alternative proof techniques in Iris. To this end, we reverify the multicopy template algorithms of [34] using our formalization of hindsight as opposed to our previous tailor-made proof argument for dealing with future-dependent linearization points. The new approach reduces the proof effort by 53%.

To summarize, our contributions are (i) template algorithms for a wide variety of lock-free search structure algorithms, (ii) mechanized proofs of linearizability based on hindsight reasoning in Iris. The result is, to our knowledge, the first formal verification of fully-functional lock-free algorithms for skip lists of unbounded height.

2 The Skip List Template Algorithm

A *skip list* is a search structure over a totally ordered set of keys \mathbb{K} . We focus our discussion on skip lists that implement mutable sets rather than maps. The extension of the presented algorithms to mutable maps is straightforward. The data structure is composed of sorted lists at multiple levels, with the base list determining the actual contents of the structure, while higher level lists are used to speed up the search. An example is shown in Figure 1. A

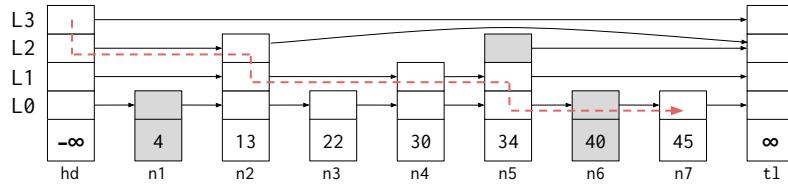


Figure 1 Skiplist with four levels. A node that is marked (logically deleted) at a level is shaded gray at that level. The red line indicates the path taken by a traversal searching for key 42.

skiplist node contains a key and has a height, determining how many higher level lists this node is a part of. Each node has a next pointer for each of its levels. Two sentinel nodes signify the head (*hd* with key $-\infty$) and the tail (*tl* with key ∞) of the skiplist. Lock-free linked lists often use the technique of logical deletion by *marking* a node before it is physically unlinked from the list. This involves storing a mark bit together with the next pointer, so as to allow reading and updating them together in a single (logically) atomic step. Lock-free skiplist implementations also use this technique. Since a skiplist node can be part of multiple lists, it has one mark bit per level.

The traversal for a key not only goes left to right as usual, but also top to bottom. The red line in Figure 1 depicts a traversal searching for key 42. The traversal begins at the highest level of the head node. At each non-base level, the traversal continues till it reaches a node with a key greater than or equal to the search key. Thereafter, the traversal drops down a level, and continues at the lower levels until it terminates on the bottom level at the first node whose key is greater than or equal to the search key.

The traversals in a concurrent skiplist perform *maintenance* in the form of physically unlinking encountered marked nodes. In Figure 1, node *n*₅ has been unlinked at level 2, thus the traversal does not visit it at that level. Operations that mark and change the next pointers at the higher levels do not affect the actual contents of the structure. We therefore consider them to be part of the maintenance.

Many variants of lock-free skiplist algorithms have been proposed in the literature and implemented in practice. These variants differ in (i) their node implementations, (ii) the styles of maintenance operations and/or (iii) the orders in which they perform maintenance operations with regard to other operations.

For example, node implementations in low-level languages often use bit-stealing [10] (or an equivalent of Java’s `AtomicMarkableReference`) so that both the next pointer and mark bit can be atomically read or updated. Other implementations use more complex solutions. For instance, the skiplists in [8] use nodes with back links to reduce traversal restarts due to marked nodes. Java’s `ConcurrentSkipListMap` [33] implements each node as a list of simpler nodes, one per level. The higher level nodes have both right pointers and down pointers, while the base nodes only have right pointers. Java’s implementation also uses *marker nodes* for marking, instead of bit-stealing.

In terms of style of maintenance, the traversal in the Michael Set [31] and Herlihy-Shavit lock-free skiplist [10, § 14] unlinks one marked node at a time. By contrast, the traversal in the Harris List [9] unlinks the entire sequence of marked nodes in one shot with a single CAS operation. The variants also differ in the order of marking of a node at higher levels. In the Herlihy-Shavit skiplist, the marking of a node goes from top level to the bottom level. This differs from skiplists in [33] and [8], whose marking goes from bottom to top.

Despite the differences in the skiplist algorithms described above (and others to be invented in the future), the bulk of their correctness reasoning remains the same. A goal of this paper is to show how to exploit that fact.

Template algorithm. Our template algorithm for skiplists abstracts away from node-level implementation details and the way in which traversals perform maintenance. As we shall see, the particular details regarding how the data is stored internal to the node does not affect the correctness of the core operations - `search`, `insert` and `delete`. Nor is the correctness affected by whether the traversal unlinks one marked node at a time or an entire sequence of marked nodes. We also show that the order in which maintenance operations are performed on the higher levels of the list does not matter for correctness. In summary, the template algorithm we present abstracts from: (i) node-level details; (ii) the style of unlinking marked nodes and (iii) the order of maintenance operations on higher levels.

The template algorithm is assumed to be operating on a set of nodes N that contains the two sentinel nodes head hd and tail tl . Let the maximum allowed height of a skiplist node be $L (> 1)$. Each node n is associated with (i) its key $\text{key}(n) \in \mathbb{K} = \mathbb{N} \cup \{-\infty, \infty\}$, (ii) its height $\text{height}(n) \in [1, L]$, (iii) the next pointers $\text{next}(n, i) \in N$ for each i from 0 to $\text{height}(n) - 1$, and (iv) its mark bits per level $\text{mark}(n, i) \in \{\text{true}, \text{false}\}$ for each i from 0 to $\text{height}(n) - 1$. When discussing $\text{next}(n, i)$ or $\text{mark}(n, i)$, we implicitly assume that i lies between 0 and $\text{height}(n) - 1$. We sometimes say a node n is unmarked to mean that it is unmarked at the base level, i.e., $\text{mark}(n, 0) = \text{false}$. The structural invariant maintains the following facts: $\text{key}(hd) = -\infty$, $\text{key}(tl) = \infty$, $\text{height}(hd) = \text{height}(tl) = L$, $\text{next}(tl, i) = tl$ for all i , $\text{next}(hd, L - 1) = tl$, $\text{mark}(hd, i) = \text{mark}(tl, i) = \text{false}$ for all i .

The core operations of the skiplist template are expressed using *helper functions* such as `findNext` and `markNode` that abstract from the details of the node implementation. We describe the behavior of these helper functions as and when we encounter them. The template is instantiated by implementing these functions. The helper functions are assumed to be *logically atomic*, i.e., appear to take effect in a single step during its execution.

Figure 2 shows the core operations of the skiplist template algorithm. (We omit the code for the data structure initialization as it is straightforward.) All three operations begin by allocating two arrays ps and cs via `allocArr`, each of size L and values initialized to hd and tl respectively. These arrays are then populated by the `traverse` operation as it computes the predecessor-successor pair for operation key k at each level. Intuitively, these pairs indicate where k would be inserted at each level. The template algorithm here abstracts away from the concrete `traverse` implementation. We later consider two implementations of `traverse` that differ in the way that maintenance is performed, as discussed earlier.

As far as the core operations are concerned, they rely on `traverse` to satisfy the following specification. First, it returns a triple (p, c, res) where p and c are nodes and res a Boolean such that $p = ps[0]$, $c = cs[0]$ and res is true iff k is contained in c . Second, the node c must have been unmarked at some point during the traversal; and third, for each $0 \leq i < L$, the traversal observes that $\text{key}(ps[i]) < k \leq \text{key}(cs[i])$.

Let us now describe the core operations, starting with the `search` operation. The `search` operation simply invokes the `traverse` function, whose result establishes whether k was in the structure. The `delete` operation starts similarly by invoking `traverse` and checking if the key is present in the structure. If it is, then `delete` invokes the maintenance operation `maintainanceOpDel`, which attempts to mark c at the higher levels (i.e. all levels except 0). We provide the implementation of `maintainanceOpDel` in a moment. Once `maintainanceOpDel` terminates, `delete` finally attempts to mark c via `markNode` at the base level. If marking succeeds, it terminates by invoking `traverse` (which performs the task of physically unlinking marked nodes at all levels) and returning `true`. Otherwise, a concurrent thread must have already marked c , in which case `delete` returns `false`.

```

1 let search k =
2   let ps = allocArr L hd in
3   let cs = allocArr L tl in
4   let _, _, res = traverse ps cs k in
5   res
6
7 let delete k =
8   let ps = allocArr L hd in
9   let cs = allocArr L tl in
10  let p, c, res = traverse ps cs k in
11  if not res then
12    false
13  else
14    maintainanceOp_del c;
15    match markNode 0 c with
16    | Success -> traverse ps cs k; true
17    | Failure -> false
18 let insert k =
19   let ps = allocArr L hd in
20   let cs = allocArr L tl in
21   let p, c, res = traverse ps cs k in
22   if res then
23     false
24   else
25     let h = randomNum L in
26     let e = createNode k h cs in
27     match changeNext 0 p c e with
28     | Success ->
29       maintainanceOp_ins k ps cs e; true
30     | Failure -> insert k

```

Figure 2 The template algorithm for lock-free skiplists. The template can be instantiated by providing implementations of `traverse` and the helper functions `markNode`, `createNode` and `changeNext`. The `markNode i c` attempts to mark node c at level i atomically, and fails if c has been marked already. `createNode k h cs` creates a new node e of height h containing k , and whose next pointers are set to nodes in array cs . Finally, `changeNext i p c cn` is a CAS operation attempting to change the next pointer of p from c to cn . `changeNext i p c cn` succeeds only if $\text{mark}(p, i) = \text{false}$ and $\text{next}(p, i) = c$. Other functions used here include `randomNum` to generate a random number and maintenance operations associated with `insert` and `delete`. `maintainanceOp_del` marks node c at the higher levels, while `maintainanceOp_ins` inserts a new node e at the higher levels.

The `insert` operation also begins with `traverse`. If the traversal returns *true*, then the key must already have been present. Hence, `insert` returns *false* in this case. Otherwise, a new node e is created using `createNode`. The node's height is determined randomly using `randomNum`, which generates a random number h such that $0 < h < L$. After creating a new node, the algorithm attempts to insert it into the list by calling `changeNext` at the base level (line 27). If the attempt succeeds, `insert` proceeds by invoking the maintenance operation `maintainanceOp_ins`, which also inserts the new node into the list at all higher levels. The `insert` then returns with *true*. If the `changeNext` operation fails, then the entire operation is restarted.

We now describe the maintenance operations for `insert` and `delete`, shown in Figure 3. The maintenance operations here differ from those in traditional skip list implementations in regards to the order in which maintenance is performed at higher levels. In traditional implementations, the marking of a node goes from top to bottom, while insertion of a new node goes from bottom to top. The skip list template presented here makes sure that the base level gets marked at the end and the insertion first happens at the base level, but it imposes no order on how it proceeds at higher levels. That is, when marking a node, a `delete` thread could for instance first mark odd levels, then even levels and finally the base level 0. The maintenance operations in the skip list template captures all such permutations. As our proof shows later, the order of maintenance at higher levels has no bearing on the correctness of the algorithm.

The `maintainanceOp_del` marks node c from levels 1 to $\text{height}(c)$. It begins by reading the height of c as h , and generating a permutation of $[1 \dots (h - 1)]$ stored in array pm via the `permute` function. The `maintainanceOp_del_rec` then recursively marks c in the order prescribed by pm . Note that the maintenance continues regardless of whether `markNode` succeeds or fails, because c will be marked at the end regardless.

30:6 Verifying Lock-Free Search Structure Templates

```

1 let maintainanceOp_del_rec i h pm c =
2   if i < h-1 then
3     let idx = pm[i] in
4     markNode idx c;
5     maintainanceOp_del_rec (i+1) h pm c
6   else
7     ()
8
9 let maintainanceOp_del c =
10  let h = getHeight c in
11  let pm = permute h in
12  maintainanceOp_del 0 h pm c
13 let maintainanceOp_ins_rec i h pm ps cs e =
14  if i < h-1 then
15    let idx = pm[i] in
16    let p = ps[idx] in
17    let c = cs[idx] in
18    match changeNext idx p c e with
19    | Success ->
20      maintainanceOp_ins_rec (i+1) h pm ps cs e
21    | Failure ->
22      traverse ps cs k;
23      maintainanceOp_ins_rec i h pm ps cs e
24  else
25    ()
26
27 let maintainanceOp_ins k ps cs e =
28  let h = getHeight e in
29  let pm = permute h in
30  maintainanceOp_ins 0 h pm ps cs e

```

 **Figure 3** The maintenance operations for the skip list. The `getHeight c` helper function returns $\text{height}(c)$. The `permute` function generates a permutation of $[1 \dots (h-1)]$ as an array.

The `maintainanceOp_ins` begins in the same way by reading the height, generating the permutation and invoking `maintainanceOp_ins_rec`. The `maintainanceOp_ins_rec` first collects the predecessor-successor pair at the current level from arrays `ps` and `cs`, respectively. Then it tries to insert the new node `e` using `changeNext` on predecessor node `p`. If `changeNext` succeeds, then the recursive operation continues. Otherwise, it recomputes the predecessor-successor pairs using `traverse`. After the recomputation, the insertion is retried at the same level.

We can now finally turn to the implementations of `traverse`. We consider two implementations that differ in their treatment of marked nodes. The *eager* traversal attempts to unlink every marked node it encounters, while the *lazy* traversal simply walks over the marked nodes till it reaches an unmarked node. The traversal then attempts to unlink the entire marked segment at once. The two implementations are similar in other aspects, so we discuss only the eager traversal in detail here.

The eager traversal is shown in Figure 4. The `traverse` function is implemented using mutually-recursive functions `eager_rec` and `eager_i`¹. The function `eager_rec` populates the arrays `ps` and `cs` with the predecessor-successor pair at level `i` computed by `eager_i`. The `eager_i` performs a traversal at level `i` by first reading the mark bit and next pointer of `c` using `findNext`. If `c` is found to be marked, then `eager_i` attempts to physically unlink the node using `changeNext`. In the case that `changeNext` fails (because either `p` is marked or it does not point to `c` anymore), `eager_i` simply restarts the `traverse` function. In the case of `Success` of `changeNext`, the traversal continues. If `c` is unmarked, then `traverse_i` proceeds by comparing `k` to `key(c)`. For $\text{key}(c) < k$, the traversal continues with `c` and `cn`. Otherwise, `eager_i` ends at `c`, returning (p, c, true) if $\text{key}(c) = k$ and (p, c, false) otherwise. As mentioned before, `eager_i` attempts to unlink immediately whenever a marked node is encountered.

¹ For ease of exposition, the implementation of the eager traversal shown in Figure 4 differs slightly from the version we have verified in Iris. The Iris version uses option return types instead of mutually-recursive functions in order to obtain a more modular proof of the eager traversal. We use the mutually recursive implementation here for clarity of exposition.

```

1 let eager_i i k p c =
2   match findNext i c with
3   | cn, true ->
4     match changeNext i p c cn with
5     | Success -> eager_i i k p cn
6     | Failure -> traverse ps cs k
7   | cn, false ->
8     let kc = getKey c in
9     if kc < k then
10       eager_i i k c cn
11     else
12       let res = (kc = k ? true : false) in
13       (p, c, res)
14 let eager_rec i ps cs k =
15   let p = ps[i+1] in
16   let c, _ = findNext i p in
17   let p', c', res = eager_i i k p c in
18   ps[i] <- p';
19   cs[i] <- c';
20   if i = 0 then
21     (p', c', res)
22   else
23     eager_rec (i-1) ps cs k
24 let traverse ps cs k =
25   eager_rec (L - 2) ps cs k
26

```

Figure 4 The eager traversal for the skip list template. `findNext i k c` returns a pair $(\text{next}(c, i), \text{mark}(c, i))$. The `getKey c` helper function returns $\text{key}(c)$.

3 Proof Intuition

Our goal is to show that the skip list template is linearizable. That is, we must prove that each of the core operations take effect in a single atomic step during its execution, the *linearization point*, and satisfies the sequential specification shown in Figure 5. For the skip list template, we define the abstract state $C(N)$ to be the union of the *logical contents* $C(n)$ of all nodes in N , where $C(n) := (\text{mark}(n, 0) ? \emptyset : \{\text{key}(n)\})$. In other words, the abstract state of the structure is a collection of keys contained in unmarked nodes at the base level. There are existing techniques from the literature that help us analyze the skip list

$$\Psi_{\text{op}}(k, C, C', \text{res}) := \begin{cases} C' = C \wedge (\text{res} \Leftrightarrow k \in C) & \text{op} = \text{search} \\ C' = C \cup \{k\} \wedge (\text{res} \Leftrightarrow k \notin C) & \text{op} = \text{insert} \\ C' = C \setminus \{k\} \wedge (\text{res} \Leftrightarrow k \in C) & \text{op} = \text{delete} \end{cases}$$

Figure 5 Sequential specification of a search structure. k refers to the operation key, C and C' to the abstract state before and after operation op , respectively, and res is the return value of op .

template. The two main techniques that we rely on are the *Edgeset Framework* [39] and *Hindsight Reasoning* [32, 22, 6, 7, 26, 27]. We begin by giving a brief overview of the two techniques, proceeded by the analysis of the skip list template using these techniques.

3.1 The Edgeset Framework

The Edgeset Framework provides a common terminology to capture how search operations navigate in a variety of search structures. We view each search structure as a mathematical graph whose edges are associated with an *edgeset*, a label that is a set of keys. We denote the edgeset from n to n' by $\text{es}(n, n')$, and $k \in \text{es}(n, n')$ signifies that a search for key k will proceed from node n to n' . In the context of the skip list template, we define the edgeset leaving n to be all values greater than the key in n if n is unmarked. If node n is marked, then the edgeset leaving n is the entire keyspace. Formally: $\text{es}(n, n') := (n' = \text{next}(n, 0) \wedge \text{mark}(n, 0) = \text{false} ? (\text{key}(n), \infty) : \mathbb{K})$. Note that, our definition of edgesets in the skip list template depends only on the base list, and not on higher level mark bits and next pointers.

30:8 Verifying Lock-Free Search Structure Templates

A notion defined in terms of edgesets is the *inset* of a node, denoted by $\text{inset}(n)$, signifying a set of keys for which a search will arrive at n . In order to understand the concept of inset intuitively, consider Figure 6. The inset of node n_4 is $(2, \infty)$, because, for all keys greater than 2, the search will enter n_4 . We say node n_1 is the *logical predecessor* of n_4 if it is the first unmarked predecessor of n_4 . The inset of the root is \mathbb{K} and the inset of n is the intersection of \mathbb{K} with the edgesets of all nodes between the root and n . For sorted linked lists in general, a more local notion gives the same result: the inset of an unmarked node n is $(\text{key}(n'), \infty)$, where n' is the logical predecessor of n .

In contrast to inset, we define the *outset* as the union of all its outgoing edgesets: $\text{outset}(n) := \bigcup_{n' \in N} \text{es}(n, n')$.

We can now define the *keyset* of a node n as $\text{keyset}(n) := \text{inset}(n) \setminus \text{outset}(n)$, i.e. intuitively, the set of keys for which a search enters n but never leaves. The importance of keysets is that if k is in $\text{keyset}(n)$, then k is either in the contents of n or is nowhere in the structure. In Figure 6, the keyset of n_4 is $(2, 9]$ and in general, the keyset of an unmarked node n is $(\text{keyset}(n'), \text{key}(n)]$ where n' is its logical predecessor. The keyset of a marked node is \emptyset because its outset is the set of all keys \mathbb{K} .

The technical definition of inset relies on the global data structure graph, defined as a solution to the following fixpoint equation

$$\forall n \in N. \text{inset}(n) = \text{in}(n) \cup \bigcup_{n' \in N} \text{es}(n', n) \cap \text{inset}(n')$$

where $\text{in}(n) := (n = \text{hd} ? \mathbb{K} : \emptyset)$. Thus, the inset is a global quantity and hence difficult to reason about. Fortunately, this is where the Flow Framework [20, 21, 28] comes in handy. It allows us to reason about quantities that can be expressed as a solution to a fixpoint equation (like inset) in a local manner by attaching *flow* values to the node. The framework then provides tools to track changes to the flow values that are induced by changes to the underlying graph. Our approach to encoding keysets in Iris using the Flow Framework is borrowed from [18]. We defer further details on this matter to the later sections.

As mentioned above, $\text{keyset}(n)$ intuitively is the set of all keys that n is responsible for. Consider Figure 6 again, a thread executing `search(6)` without any interference will reach node n_4 and terminate, concluding that 6 is not present in the structure. In this sense, we say n_4 is responsible for key 6 and therefore 6 is part of n_4 's keyset. The keysets of all nodes partition the set of all keys and provide the crucial *Keyset Property*:

$$\forall n \in N, k \in \mathbb{K}. k \in \text{keyset}(n) \Rightarrow (k \in C(N) \Leftrightarrow k \in C(n)) \quad (\text{KeysetPr})$$

This property enables one to lift a proof of the specification at the node level to a proof of the sequential specification Ψ_{op} . A particular situation where (KeysetPr) proves indispensable is when `search` fails to find the search key. Note that `search` observes only the nodes it visited, and hence has only a partial view of the structure. When `search` fails to find the key, the proof has to reconcile this partial view of the structure with the global view. In essence, if a concurrent invocation of `search` on key k fails to find the key, can we conclude that there was a point in time during its execution when k was in fact not present in the structure? Here, the property (KeysetPr) helps us reconcile facts gathered by `search` with the global state of the structure. Specifically, if `search` can determine a node n such that $k \in \text{keyset}(n)$ and $k \notin C(n)$, then we can immediately infer that k was not present in the structure at that point in time.

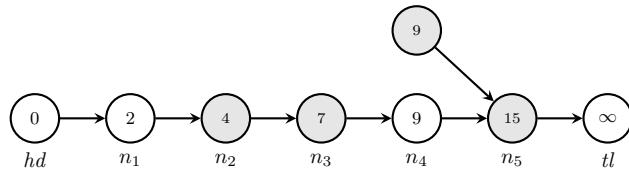


Figure 6 Possible state of the base list in the skip list template. Nodes are labeled with the value of their `key` field. Edges indicate `next` pointers. Marked (logically deleted) nodes are shaded gray. $\text{keyset}(hd) = \{0\}$, $\text{keyset}(n_1) = (0, 2]$, $\text{keyset}(n_4) = (2, 9]$ and $\text{keyset}(tl) = (9, \infty)$. The keyset of a marked node is always \emptyset .

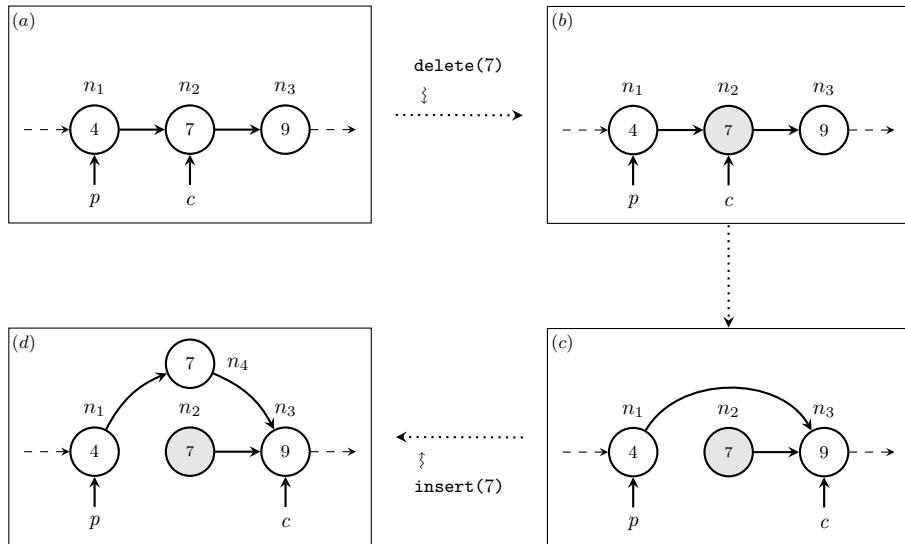


Figure 7 Possible states of `search(7)` on the base level in presence of interference from concurrent `delete(7)` and `insert(7)`.

3.2 Hindsight Reasoning

Lock-free structures often exhibit future-dependent linearization points. That is, the linearization point of an operation cannot be determined at any fixed moment, but only at the end of the execution, once any interference of other concurrent operations has been accounted for. To understand the interference issue, consider the `search` operation. Since, `search` returns the result of `traverse`, let us look at the eager traversal implementation. To simplify the explanation further, let us assume that the maximum height allowed for every non-sentinel node is one. Then, we can ignore the `eager_rec` function and focus on `eager_i` called at the base level.

Let there be a thread T executing `search(7)`. Concurrently, there is a thread T_d executing `delete(7)` and a thread T_i executing `insert(7)`. Figure 7 shows interesting scenarios that thread T might potentially observe. Box (a) captures the state of the structure at the beginning of the `eager_i` call processing n_2 . Let **Scenario 1** be the situation when thread T faces no interference from T_d and T_i . Here, thread T finds the key 7 in n_2 and `eager_i` returns `true`. The point when `eager_i` finds n_2 to be unmarked becomes the linearization point for this scenario.

Now consider **Scenario 2** to be the situation where thread T_d marks n_2 before `eager_i` processes it, as shown in Box (b). Thread T will attempt to unlink n_2 , and assuming no further interference, the unlink will result in the structure in Box (c). Thread T will process

n_3 next, finding n_3 to be unmarked with key greater than 7, and will terminate with result *false*. So when is the linearization point in this scenario? It cannot be when T finds n_3 unmarked when processing it. Because there could be further interference from thread T_i which inserts key 7 in a new node as shown in Box (d). The new node could be added right before T reads the mark bit of n_3 . Thus, when `eager_i` finds n_3 unmarked and returns *false*, key 7 could actually be present in the structure at that point in time.

The linearization point is actually the point in time shown in Box (c), i.e., right after n_2 is unlinked. However, thread T cannot confirm this when n_2 is unlinked because `eager_i` may not terminate at n_3 with *false* as the result. The reason is that by the time T processes n_3 , it could get marked in a manner similar to n_2 in Box (b), resulting in the unlinking of n_3 and potentially a restart. That Box (c) is the linearization point is confirmed when T has found n_3 to be unmarked later. The structure maintains the invariant that once a node is marked, it remains marked. Using this invariant, an analysis of thread T 's history concludes that n_3 must have been unmarked at the point when n_2 was unlinked. Once `eager_i` terminates at n_3 with *false*, an analysis can *establish in hindsight* that Box (c) indeed was the linearization point.

Hindsight reasoning as formalized in [26, 27] is designed to deal with situations like the `search` in Figure 7. It enables temporal reasoning about computations using a *past predicate* $\diamond q$, which expresses that proposition q held true at some prior state in the computation (up to the current state). For instance, $\diamond(\text{next}(n_1, 0) = n_2)$ holds in Box (c) even though $\text{next}(n_1, 0) = n_3$ at that point. The reason is that $\text{next}(n_1, 0) = n_2$ was true at an earlier point in time, namely in Box (b). Note that the past operator \diamond abstracts away the exact time point when the predicate held true. Note also that a past predicate is not affected by concurrent interferences, as it merely records some fact about a past state.

There are two ways to establish a past predicate that are relevant for our proofs. The first is to establish the predicate in the current state directly. That is, $\diamond q$ holds if q holds in the current state. As an example, we obtain $(\text{next}(n_1, 0) = n_2)$ when `findNext` on n_1 returned n_2 in Box (a). Thus, for all subsequent states including Box (b) and (c), we get $\diamond(\text{next}(n_1, 0) = n_2)$. The second way to establish a past predicate is through the use of *temporal interpolation* [27]. That is, one proves a lemma of the form: if there existed a past state that satisfied property q and the current state satisfies r , then there must have existed an intermediate state that satisfied o . Such lemmas can then be applied, e.g., to prove that if thread T finds n_3 to be unmarked in Scenario 2, then it must have been unmarked when n_2 was unlinked in Box (c).

Equipped with the Edgeset Framework and hindsight reasoning, we are now ready to analyze the core operations of the skip list template.

3.3 Proof Outline for Core Operations

We refer to a linearization point as *modifying* if the operation changes the abstract state of the data structure (like in the case of a succeeding `delete` and `insert`) and otherwise refer to it as *unmodifying* (like `search` and in the case of a failing `delete` or `insert`). The modifying linearization points of the skip list template are easier to reason about because they are not future-dependent. For `delete`, the linearization point occurs when `markNode` succeeds, and similarly, for `insert` the linearization point occurs when the call to `changeNext` on line 27 succeeds. The proof strategy for unmodifying linearization points is to combine (`KeysetPr`) with the \diamond operator from hindsight reasoning. Let us expand on this proof strategy in detail and show why the skip list template is linearizable.

We begin by describing the specification for `traverse` that is assumed for analyzing the core operations of the template. Then, we analyze each of the operations in detail. Finally, we show how the eager implementations of `traverse` satisfies the specification that was assumed in the beginning. Along the way, we introduce (as and when necessary) invariants maintained by the skiplist template that are crucial for proving linearizability.

Specification of `traverse`. The function `traverse ps cs k` updates arrays `ps` and `cs` with predecessor-successor pairs for each level and returns a triple (p, c, res) that satisfies the following past predicate regarding node c : $\Diamond(k \in \text{keyset}(c) \wedge (res \Leftrightarrow k \in C(c)))$. Recall that our definition of edgesets in Section 3.1 implies the following invariant:

Invariant 1 For all nodes n , if $\text{mark}(n, 0)$ is set to *true* then $\text{keyset}(n) = \emptyset$.

Using Invariant 1, we can establish that c is unmarked at the base level at the time point when $k \in \text{keyset}(c)$ holds. Note that `traverse` may physically unlink marked nodes. However, this step does not change the abstract state of the structure. Hence, the specification for `traverse` involves no change of the abstract state.

We now consider each of the core operations in detail.

Proof of search. Function `search` returns `res` out of the triple (p, c, res) returned by `traverse`. The specification of `traverse` says $res \Leftrightarrow k \in C(c)$ at some point, say t , during its execution. The specification additionally guarantees $k \in \text{keyset}(c)$ at time t . These two facts, combined with the (`KeysetPr`) at time point t , allow us to immediately infer that `res` is *true* iff k was in the structure at that point. Hence, we can establish that $(res \Leftrightarrow k \in C(c))$ was true at some point during the execution of `search`.

Proof of delete. We analyze `delete` by case analysis on the value `res` returned by `traverse`. If `res` is *false*, then again we can establish that k was not in the structure at some point during `traverse`'s execution by the same reasoning used in the proof of `search`. So let us consider the case that `res` is *true*. By the specification of `traverse`, we can establish a time point when c was unmarked and contained k . The `delete` operation then calls `maintainanceOpDel` which marks c at all the higher levels. Finally, the `markNode` on Line 15 attempts to mark c at the base level. If `markNode` succeeds, then this step becomes the linearization point of `delete` and k can be considered to be deleted from the structure. But if `markNode` fails, then we gain the knowledge that $\text{mark}(c, 0) = \text{true}$. Hindsight reasoning allows us to infer that c was marked at the base level by a concurrent thread between the end of `traverse` and the invocation of `markNode`. The point right after c was marked by a concurrent thread becomes the linearization point of `delete` in this case, as we can determine that k was not present in the structure at that point.

This hindsight reasoning relies on two facts: first, the key of a node never changes and second, once a mark bit is set to *true* by a successful `markNode` operation (at line 15 in `delete` or line 4 in `maintainanceOpDel`), no other operation will set it back to *false*. In fact, these two facts are invariant for the skiplist template:

Invariant 2 For all nodes n and level i , once $\text{mark}(n, i)$ is set to *true*, it remains *true*.

Invariant 3 For all nodes n , $\text{key}(n)$ remains constant.

Proof of `insert`. Similar to `delete`, we begin by case analysis on `res` returned by `traverse`. If `res` is true, then we can establish that k was already present in the structure at some point. Otherwise, `res` is `false` and `insert` creates a new node e with key k . Using `changeNext`, an attempt is made to insert node e between nodes p and c . If the attempt succeeds, then k is now part of the structure and this becomes the linearization point. The following `maintainanceOp_ins` operation does not change the abstract state of the structure, and thus, has no effect in terms of linearizability. If the `changeNext` fails, then `insert` simply restarts.

As is evident with the proof outline for the core operations, the specification assumed for `traverse` plays a critical role in case the operation exhibits an unmodifying linearization point. Let us now turn to `traverse` and show how its specification can be proved. We analyze the eager traversal in detail in the following section. The proof argument for the lazy version is similar.

3.4 Proof Outline for Eager Traversal

As stated earlier, `traverse` returns (p, c, res) such that $\diamondsuit(k \in \text{keyset}(c) \wedge (res \Leftrightarrow k \in C(c)))$. Since the returned triple is the result of a call to `eager_i` at the base level, let us begin by analyzing the behavior of this call.

In the sequential setting, the traversal in a search structure maintains the invariant that the search key is always in the inset of the current node. This invariant holds by the design of the Edgeset Framework. Unfortunately, this invariant no longer holds for the skiplist template in the concurrent setting as evidenced by Box (c) in Figure 7. However, we argue first that `eager_i` does maintain the invariant that the search key was in the inset of the current node c between the start of the traversal and the point at which the `eager_i` accesses c . We call this the *inset in hindsight* invariant.

We prove this invariant inductively. We make use of the following locally maintained invariants: (i) At all times, there is one list, denoted the *reachable list*, from the head node that includes all unmarked and some marked nodes. (This list is characterized by the set of nodes with non-empty inset, see Figure 6 for intuition). (ii) The keys in the reachable list are sorted. A consequence of these two invariants is that if a node n is in the reachable list (whether n is marked or not) and has a key less than k , then k is in the inset of n .

To prove that *inset in hindsight* is an invariant, we have to show that (a) it is an invariant when `eager_i` takes a step (Line 2) when traversing the base level, and (b) that we can establish *inset in hindsight* when `eager_rec` initiates `eager_i` (Line 17) at the base level.

To show (a), observe that if a node n becomes unlinked from the reachable list, then it will never again be part of the reachable list. Hence, if n is not in the reachable list when `eager_i` begins executing at the base list, then `eager_i` will never visit n . The contrapositive of this statement allows us to say that if `eager_i` reaches some node c , then it must have been part of the reachable list at some point during the execution of `eager_i`. Additionally, `eager_i` proceeds to the node following c only when $\text{key}(c) < k$. With the help of invariants (i) and (ii) above, we can thus establish that k was in the inset of n at some point.

To show (b), we must do a case analysis on whether node p (Line 16) is marked. If it is unmarked, then it is straightforward to establish that k is in the inset of c currently. However, if p is marked, then we require temporal interpolation based on the following invariant:

Invariant 4 For all nodes n and level i , once `mark(n, i)` is set to `true`, `next(n, i)` does not change.

This invariant tells us that if p was known to be unmarked in the past, and it is marked currently, then p must have been pointing to c right before it got marked. At that point in time, we can establish that k must have been in the inset of c .

This completes the inductive proof that inset in hindsight is indeed an invariant maintained by the traversal. The inset in hindsight invariant is sufficient to prove the `traverse` specification by the following simple argument. If the traverse encounters k in an unmarked node n , then `traverse` will return *true* as it should. If, by contrast, `traverse` encounters an unmarked node n such that $\text{key}(n) > k$, then by the inset in hindsight invariant, k must have been in the inset of n at some point t in the past and k cannot be in the outset of n (because $\text{key}(n) > k$ and n is unmarked), so therefore k must have been in the keyset of n at time t .

4 Hindsight Reasoning in Iris

Linearizability in Iris is defined via (*logically*) *atomic triples* [4, 16]. Intuitively, an atomic triple $\langle x. P \rangle e \langle v. Q \rangle$ says that at some point during the execution of e , the resources described by the precondition P will be updated to satisfy the postcondition Q for return value v in one atomic step. The variable x can be thought of as the abstract state of the data structure before the update at the linearization point.

Linearizability of a search structure operation op can be expressed by an atomic triple of the form

$$\boxed{\text{Inv}(r)} \dashv\ast \langle C. \text{CSS}(r, C) \rangle \text{op} r k \langle \text{res}. \exists C'. \text{CSS}(r, C') * \Psi_{\text{op}}(k, C, C', \text{res}) \rangle. \quad (\text{ClientSpec})$$

Here, r is the pointer to the head of the data structure. The predicate $\text{CSS}(r, C)$ is the *representation predicate* that relates the head pointer with the contents C of the structure. The predicate $\text{Inv}(r)$ is the shared data structure invariant. It can be thought of as a thread-local precondition of the atomic triple, which we express using separating implication. The invariant ties $\text{CSS}(r, C)$ to the data structure's physical representation and may contain other resources necessary for proving the atomic triple. The predicate $\Psi_{\text{op}}(k, C, C', \text{res})$ captures the sequential specification of the structure. The specification essentially says there is a single atomic step in op where the abstract state changes from C to C' according to the sequential specification $\Psi_{\text{op}}(k, C, C', \text{res})$ (Figure 5). This step is op 's linearization point. We call (ClientSpec) the *client-level* atomic specification for the data structure under proof.

Proving atomic triples. The proof of establishing an atomic triple involves a *linearizability obligation* that must be discharged directly at the linearization point. However, it can be challenging to determine the linearization point precisely and to discharge the linearizability obligation exactly at that point. When the program execution reaches a potential linearization point that depends on future interferences by other threads, then the proof will fail if it is unable to determine whether the linearizability obligation should be discharged now or later. In Iris, this challenge is overcome using *prophecy variables* [15], which enable the proof to reason about the remainder of the computation that has not yet been executed.

Another challenge is that the linearization point of an operation may be an atomic step of another operation that is executed by a different thread (like in Scenario 2 discussed in Section 3.2). Data structures that demonstrate such behavior are said to deploy *helping*. This behavior complicates thread modular reasoning. The conventional solution to this challenge in Iris is to use a *helping protocol* [15, 34, 13]. The helping protocol is specified as part of the shared data structure invariant and consists of a registry that tracks which threads are expected to be linearized by other threads as well as conditional logic that governs the correct transfer and discharge of the associated linearizability obligations.

Both the use of prophecy variables and the helping protocol need to be tailored to the specific data structure at hand, which adds considerable overhead to the proof. To reduce this overhead, we present an alternative proof method that enables linearizability proofs based on hindsight arguments in Iris. Rather than identifying the linearization point precisely, the proof can establish linearizability in hindsight using temporal interpolation in the style of the intuitive proof argument for the skiplist template presented in Section 3.2.

Hindsight specification. Our proof method offers an intermediate specification, a Hoare triple specification, which in essence expresses that linearizability has been established in hindsight. In our Iris formalization, we show that any data structure whose operations satisfy the hindsight specification also satisfy the client-level atomic specification. This proof relates the two specifications via prophecy variables and a helping protocol. However, the helping protocol is data structure agnostic, making our proof method applicable to a broad class of structures exhibiting future-dependent unmodifying linearization points.

From the perspective of a proof author using our method to prove linearizability of some structure, one has to only establish the hindsight specification to obtain the proof of the client-level atomic specification. To this end, our method provides further guidance to the proof author.

In order to use hindsight reasoning, one has to have the history of computation at hand. Here, we offer a shared state invariant with a mechanism to store the history. The shared state invariant has three main components: a mechanism to store the history, the helping protocol, and finally, an abstract predicate that can be instantiated with invariants specific to the structure at hand. The first two components are data structure agnostic. The proof author only needs to specify the data structure-specific invariant and what information about the data structure state should be tracked by the history.

In the rest of this section, we discuss our method in detail. We begin with the hindsight specification, followed by a discussion of the shared state invariant and how to use it.

4.1 Linearizability in Hindsight

We motivate the hindsight specification using the challenges we face when proving the client-level atomic specification for the `delete` operation of the skiplist template. Let us recall the intuitive proof argument for `delete` from Section 3.3. As per the observation regarding the modifying and unmodifying linearization points, a `delete` thread with modifying linearization point can fulfill the obligation at the point when the structure is modified. However, a `delete` thread with an unmodifying linearization point requires helping.

Prophecy reasoning. An important detail of our proof method is how it determines whether a thread requires helping. In the following, we refer to the operation that a thread performs at its linearization point as its *decisive operation*. In `delete`, the traversal observes node c to be unmarked at some point during execution. In the case where c is marked by the time that the thread calls its decisive operation `markNode` (in Line 15), the thread requires helping from the thread that marks c .

In order to determine in advance whether a thread requires helping, our proof method attaches a prophecy to each thread. A prophecy in Iris can predict a sequence of values and is treated as a resource that can be owned by a thread. Ownership of a prophecy p is captured by the predicate `Proph(p, pvs)`, where pvs is the list of predicted values. The predicate signifies the right to resolve p when the thread makes a physical step that produces some result value v . The resolution of p establishes equality between v and the head of the

list pvs (i.e., the next value predicted by p). The resolution step yields the updated predicate $\text{Prop}(p, pvs')$ where pvs' is the tail of pvs . This mechanism enables the proof to do a case analysis on the predicted values pvs before these values have been observed in the program execution².

The prophecy attached to a thread predicts the results of the thread's decisive operation. In case of `delete`, the decisive operation is the call to `markNode` in the base list, while for `insert`, it is the call to `changeNext` in the base list. Note that a thread may restart if its decisive operation fails (like in the case of `insert`). Therefore, the prophecy needs to predict a sequence of result values, one for each attempted call to the thread's decisive operation.

For the purpose of this discussion, we assume that the prophecy predicts a sequence of **Success** or **Failure** values. If the sequence contains a **Success** value, then the decisive operation will succeed and the thread will modify the structure. Otherwise, the thread's linearization point is unmodifying. Let predicate $\text{Upd}(pvs)$ hold when pvs contains at least one **Success** value.

The proof author only needs to identify the decisive operations that potentially change the abstract state of the structure (like `markNode` as discussed above) by resolving the prophecy around these decisive calls.

Hindsight specification. Before we can present the hindsight specification, we need to provide necessary details regarding the atomic triples in Iris. An atomic triple $\langle x.P \rangle e \langle v.Q \rangle$ is defined in terms of standard Hoare triples of the form $\forall \Phi. \{AU_{x,P,Q}(\Phi)\} e \{v.\Phi(v)\}$. The predicate $AU_{x,P,Q}(\Phi)$ is the *atomic update token* and represents the linearizability obligation of the atomic triple. At the beginning of each atomic step that the thread takes up to its linearization point, the token offers the resources in P and the token itself transforms into a choice. That is, at the end of the atomic step, the prover has to choose to either *commit* the linearization or *abort*. When committing, the prover has to show that the thread's atomic step transforms the resources in P to those in Q , receiving $\Phi(v)$ from the update token in return, which serves as the receipt of linearization of the atomic triple. In case of an abort, the prover needs to show that the thread's atomic step reestablishes P .

We also need to introduce two more auxiliary predicates:

- $\text{Thread}(tid, t_0)$: this predicate is used to *register* the thread with identifier tid in the shared invariant. The argument t_0 denotes the time when thread tid began its execution.
 - $\text{PastLin}(op, k, res, t_0)$: this predicate holds if there was a past state in the history between time t_0 and the point when this predicate is evaluated for which the sequential specification Ψ_{op} held with result res . It essentially captures whether the sequential specification was true for any point after time t_0 .

We now have all the ingredients to present the hindsight specification:

$$\begin{aligned} & \forall tid t_0 pvs. \boxed{\text{Inv}(r)} \rightarrow \text{Thread}(tid, t_0) \rightarrow \\ & \left\{ \begin{array}{l} \{\text{Proph}(p, pvs) * (\text{Upd}(pvs) \rightarrow \text{AU}_{\text{op}}(\Phi))\} \text{ op } r k \\ \left\{ \begin{array}{l} \text{res. } \exists pvs'. \text{Proph}(p, pvs') * pvs = (_\circledast pvs') \\ * (\text{Upd}(pvs) \rightarrow \Phi(\text{res})) \end{array} \right\} \\ * (\neg \text{Upd}(pvs) \rightarrow \text{PastLin}(\text{op}, k, \text{res}, t_0)) \end{array} \right\} \end{aligned} \quad (\text{HindSpec})$$

² For further details on prophecies in Iris, we refer to [15].

We explain it piece by piece. The local precondition $\text{Thread}(tid, t_0)$ ties the thread to its identifier tid and provides knowledge that tid begins executing at time t_0 . The Hoare triple can be best understood by observing how prophecy resources are allowed to change (highlighted in brown) and what are the obligations when $\text{Upd}(pvs)$ holds (in teal) versus when it does not hold (in magenta). Let us look at each of these in detail. First, the prophecy resource $\text{Proph}(p, pvs)$ in the precondition changes to $\text{Proph}(p, pvs')$ in the postcondition where pvs' is a suffix of pvs . It basically says that operation op is allowed to resolve the prophecy p as many times as it needs and then return the remaining resource at the end.

Now let us consider the case when $\text{Upd}(pvs)$ holds. The precondition here provides the atomic update token $\text{AU}_{op}(\Phi)$ to op , expecting the receipt of linearization $\Phi(res)$ in return. Thus, the responsibility of linearization is delegated to op when $\text{Upd}(pvs)$ holds. We can gain better insight by relating this situation to the `delete` operation from the skip list template as before. This case corresponds to when `markNode` (from line 15) succeeds as $\text{Upd}(pvs)$ holds here. The point when `markNode` succeeds becomes the linearization point and so the thread does not require help from other threads to linearize. The hindsight specification simply asks for the receipt from linearization $\Phi(res)$ at the end.

Finally, let us consider the case when $\text{Upd}(pvs)$ does not hold. The precondition provides no additional resources here, while the postcondition requires the predicate $\text{PastLin}(op, k, res, t_0)$. In simple terms, this means that if $\text{Upd}(pvs)$ is not true, i.e., the prophecy says the thread is not going to modify the structure, then the hindsight specification allows exhibiting a past state from history when the sequential specification was true. Relating again to `delete`, if the `markNode` fails, then the thread can look at the history of the structure and exhibit precisely the point when the decisive node got marked.

The proof argument for establishing the hindsight specification is significantly simpler than if one were to attempt a direct proof of the client-level atomic specification. In particular, the proof author does not need to reason about helping and atomic update tokens in last case discussed above. Instead, they only need to reason about the structure-specific history invariant.

Soundness of the hindsight specification. Our proof that relates the hindsight specification for op to the atomic triple specification involves a helping protocol. The details of the helping protocol and the soundness proof for the hindsight specification are similar to those of the proofs presented in [15, 34]. We therefore provide only a brief summary here. Additional details regarding the proof and the helping protocol can be found in [36].

Before op begins executing, the proof creates the prophecy resource $\text{Proph}(p, pvs)$ assumed in the precondition of the hindsight specification. If the prophecy determines that the thread requires helping, then its client-level atomic triple is registered to a predicate which encodes the helping protocol as part of the shared state invariant $\text{Inv}(r)$. The registered atomic triple serves as an obligation for the helping thread to commit the atomic triple. This obligation will be discharged by the appropriate concurrent operation determined by the op 's sequential specification Ψ_{op} . The proof then uses the hindsight specification to conclude that it can collect the committed triple from the shared predicate. The committed triple serves as a receipt that the obligation to linearize has been fulfilled.

To govern the transfer of linearizability obligations and fulfillment receipts between threads via the shared invariant, the helping protocol tracks a *registry* of thread IDs with unmodifying linearization points that require helping from other concurrent threads. Each thread registered for helping is in either *pending* state or *done* state, depending on whether the thread has already been linearized. A thread registered for helping must be able to

determine its current protocol state in order to be able to extract its committed atomic triple from the registry. For this purpose, the helping protocol includes a *linearization condition* that holds iff a registered thread *tid* has linearized (and is, hence, in *done* state).

From the point of view of a thread which *does* the helping, the linearization condition forces its proof to scan over the pool of uncommitted triples registered in the helping protocol and identify those that need to be linearized at its linearization point, changing their protocol state from *pending* to *done*. This step involves a proof obligation for the helping thread to show that the sequential specification of *tid*'s operation is indeed satisfied at the linearization point.

One crucial innovation in our helping protocol is that we have formulated a linearization condition that is parametric in the sequential specification of the data structure operations, making the soundness proof for the hindsight specification applicable to many structures at once. In particular, we deal with the aspect of scanning and updating the registry in the proof of the helping thread, the proof author simply invokes a lemma provided by our method at the identified linearization points. Therefore, the helping protocol mechanism remains fully opaque to the proof author.

4.2 Invariant for Hindsight Reasoning

Hindsight arguments involve reasoning about past program states. Our encoding therefore tracks information about past states using *computation histories*. We define computation histories as finite partial maps from *timestamps*, \mathbb{N} , to *snapshots*, \mathbb{S} . A snapshot describes an abstract view of a program state. It is a parameter of our method. For instance, a snapshot may capture the physical memory representation of the data structure under proof, while abstracting from the remainder of the program state. Another parameter is a function $|\cdot|$ that computes the abstract state of the data structure from a given snapshot.

$$\begin{aligned} \text{Inv}(r) &:= \exists H T C. \overline{\text{CSS}}(r, C) * |H(T)| = C \\ &\quad * \text{Hist}(H, T) * \text{Inv}_{\text{help}}(H, T) * \text{Inv}_{\text{tpl}}(r, H, T) \\ \text{Inv}_{\text{tpl}}(r, H, T) &:= \text{resources}(r, H(T)) \\ &\quad * (\forall t, 0 \leq t \leq T \Rightarrow \text{per_snapshot}(H(t))) \\ &\quad * (\forall t, 0 \leq t < T \Rightarrow \text{transition_inv}(H(t), H(t + 1))) \end{aligned}$$

Figure 8 Definition of the shared state invariant encoding the hindsight reasoning. Variable H represents the history, T the current timestamp in use and C the abstract state of the structure.

Figure 8 shows a simplified definition of the invariant that encodes the hindsight reasoning. For sake of brevity, we provide only a high-level overview of the predicates used in the invariant. The predicate $\text{Hist}(H, T)$ contains the mechanism to track the history of snapshots. That is, H denotes the history that has been observed so far and T is the current time stamp. Using appropriate ghost resources, it ensures that the timestamps are non-decreasing and past states recorded in H are preserved by future updates to the history. This allows us to define a *past predicate* $\diamondsuit_{s,t_0}(q)$ with the intuitive meaning that the history contains state s recorded after (or at) time t_0 for which proposition q holds true. The exact definition of the past predicate uses the ghost resources used to preserve the past states. The predicate $\text{Hist}(H, T)$ also guarantees that $\text{dom}(H) = \{0 \dots T\}$, ensuring that there are no gaps in the history.

The conjunct $|H(T)| = C$ and the predicate $\overline{\text{CSS}}(r, C)$ together tie the abstract state C of the data structure to the latest snapshot in the history. The predicate $\overline{\text{CSS}}(r, C)$ is the dual of the representation predicate $\text{CSS}(r, C)$ used in the client-level atomic specification. Both represent one half of an ownership over the abstract state of the structure, keeping the abstract state defined by $\text{Inv}(r)$ synchronized with the representation predicate $\text{CSS}(r, C)$.

The helping protocol predicate $\text{Inv}_{\text{help}}(M, T)$ contains a *registry* of thread IDs with unmodifying linearization points that require helping from other concurrent threads. For each thread ID tid in the registry, the protocol stores information such as the start time of the thread, whether it has been linearized or not, etc.

The predicate $\text{Inv}_{\text{tpl}}(r, H, T)$ captures invariants particular to the data structure under proof. It is further composed of three abstract predicates that are meant to be instantiated with the structure specific invariants. The three predicates serve the following purpose. The first predicate $\text{resources}(r, H(T))$ ties the current snapshot to the physical representation of the structure. The predicate $\text{Hist}(H, T)$ contains a conjunct $(\forall t, t < T \Rightarrow H(t) \neq H(t+1))$. Together with the predicate resources , this conjunct forces a thread to update the history whenever the structure is modified.

The predicate $\text{per_snapshot}(H(T))$ captures the structural invariants that hold for any given snapshot. For instance, when proving the skip list template, this predicate holds facts about the nodes hd and tl having maximum height, etc. The predicate $\text{transition_inv}(s, s')$ captures a transition invariant on snapshots observed in the history. That is, it constrains how certain quantities evolve over time. Again as an example from the skip list template proof, the fact that a node marked in s remains marked in s' is included here. Crucially, the facts in $\text{transition_inv}(s, s')$ allow temporal interpolation required to establish facts about past states in the history (like in Section 3.2).

To summarize, the proof author defines the snapshot of the structure, the function $|\cdot|$, and instantiates the three abstract predicates in Inv_{tpl} appropriately. The resulting shared state invariant then tracks the history and handles the helping protocol without requiring further fine-tuning to the data structure at hand.

5 Verifying the Skip List Template

We relate the intuitive proof argument from Section 3 to the development on hindsight reasoning in Iris in Section 4 to obtain a complete proof of the skip list template. To achieve this, we must perform three tasks required by the proof method in Section 4. The first task is to determine the decisive operations that potentially alter the structure, and resolve the prophecy around those operations. As discussed previously, the decisive operations are `markNode` for `delete` and `changeNext` for `insert`. The `search` operation does not modify the abstract state and hence, it has no decisive operation.

The second task is to define a snapshot in the context of the skip list template and instantiate Inv_{tpl} appropriately. This includes the predicate resources that ties the concrete state of the structure to the latest snapshot, as well as invariants that allow temporal interpolation. The third and the final task is to prove the hindsight specification for the core operations.

In this section we focus on the second task of defining the snapshot and providing invariants necessary to formalize the intuitive proof argument. Once, we have set up the right invariants, the formalized proof follows the intuitive proof very closely. We explain this with `delete` as an example.

```

 $\text{Inv}_{\text{tpl}}(r, H, T) := \text{resources}(r, H(T))$ 
    *  $(\forall t, 0 \leq t \leq T \Rightarrow \text{per\_snapshot}(H(t)))$ 
    *  $(\forall t, 0 \leq t < T \Rightarrow \text{transition\_inv}(H(t), H(t + 1)))$ 
 $\text{resources}(s) := \bigstar_{n \in \text{FP}(s)} \text{Node}(n, \text{mark}(s, n), \text{next}(s, n), \text{key}(s, n), \text{height}(s, n))$ 
    *  $\text{resources\_keyset}(s)$ 
 $\text{transition\_inv}(s, s') := (\text{FP}(s) \subseteq \text{FP}(s'))$ 
    *  $(\forall n, \text{key}(s', n) = \text{key}(s, n) \wedge \text{height}(s', n) = \text{height}(s, n))$ 
    *  $(\forall n i, \text{mark}(s, n, i) = \text{true} \Rightarrow \text{mark}(s', n, i) = \text{true})$ 
    *  $(\forall n i, \text{mark}(s, n, i) = \text{true} \Rightarrow \text{next}(s', n, i) = \text{next}(s, n, i))$ 

```

■ **Figure 9** Instantiating Inv_{tpl} with invariants of the skip list template.

5.1 Snapshot and the Skip list Template Invariant

Recall that the notion of keysets are central to the intuitive proof argument for the core operations of the skip list template. Hence, a snapshot of the structure must contain information about the keysets. For encoding keysets in Iris, we borrow heavily from [18], especially the *keyset camera* and the representation of keysets via the Flow Framework.

We define the snapshot of the skip list template as a tuple containing the following components:

- the set of nodes N comprising the structure (also referred to as the *footprint* below)
- the abstract state of the structure (a set of keys)
- the mark bits (a map from N to $\mathbb{N} \rightarrow \text{Bool}$, i.e., a Boolean per level)
- the next pointers (a map from N to $\mathbb{N} \rightarrow N$)
- the keys (a map from N to K)
- the height of nodes (a map from N to \mathbb{N})
- the representation of flow values

We reparameterize the $\text{mark}(n, i)$ function introduced earlier to take the snapshot as an argument. Thus, we use $\text{mark}(s, n, i)$ to mean the mark bit of node n at level i in snapshot s . We redefine $\text{next}(\cdot)$, $\text{key}(\cdot)$, $\text{keyset}(\cdot)$ and other such functions similarly by adding the snapshot s as an additional parameter. We also use $\text{FP}(s)$ to represent the footprint of the snapshot s .

We now present the skip list template invariant in Figure 9. The `resources` predicate ties the snapshot to the concrete state through an intermediary node-level predicate `Node`. This predicate actually ties the physical representation of a node in the heap to the abstract quantities ($\text{key}(\cdot)$, $\text{height}(\cdot)$, $\text{mark}(\cdot)$ and $\text{next}(\cdot)$, respectively) that the skip list template relies on. The `Node` predicate also owns all the resources needed to execute the helper functions. The skip list template proof is parametric in the definition of `Node`. Thus, we achieve proof reuse across skip list variants that follow the same high-level skip list algorithm, but implement the node differently. We provide more details on this matter later. We discuss some concrete node implementations in Section 6.

The predicate `resources_keyset` captures the ownership resources required for keyset reasoning. Using the ghost resources in Iris and the keyset camera from [18], it ensures that the keysets and the logical contents of nodes in s satisfy (`KeysetPr`).

```

1 〈 k h mk nx. Node(n, k, h, mk, nx) 〉 getKey n 〈 k. Node(n, k, h, mk, nx) 〉
2 〈 k h mk nx. Node(n, k, h, mk, nx) 〉 getHeight n 〈 h. Node(n, k, h, mk, nx) 〉
3 〈 k h mk nx. Node(n, k, h, mk, nx) * (i < h) 〉 findNext i n 〈 n'. Node(n, k, h, mk, nx) * (nx(i) = n') 〉
4
5 〈 k h mk nx. Node(n, k, h, mk, nx) * (i < h) 〉 markNode i n
6 〈 x. Node(n, k, h, mk', nx) * (mk(i) = true ⇒ x = Failure * mk' = mk) 〉
7 〈 x. Node(n, k, h, mk', nx) * (mk(i) = false ⇒ x = Success * mk' = mk[i ↦ true]) 〉
8 〈 k h mk nx. Node(n, k, h, mk, nx) * (i < h) 〉 changeNext i n n' e
9 〈 x. Node(n, k, h, mk, nx') * ((mk(i) = true ∨ nx(i) ≠ n') ⇒ x = Failure * nx' = nx) 〉
   〈 ((mk(i) = false ∧ nx(i) = n') ⇒ x = Success * nx' = nx[i ↦ e]) 〉

```

Figure 10 Specifications of the helper functions used by the skiplist template.

The predicate `per_snapshot` captures structural invariants that hold for all snapshots recorded in the history. This includes invariants of three kinds: first, invariants to ensure that each component of the snapshot is of the correct type and the maps (from nodes to mark bits, next pointers, etc.) are defined for all nodes in the footprint; second, the node-level invariants relating the node's inset, outset, mark bit, etc (like Invariant 1); and third, invariants about the `hd` and `tl` nodes, such as $\text{key}(s, \text{hd}) = -\infty$, $\text{height}(\text{tl}) = L$, etc.

The predicate `transition_inv(s, s')` captures invariants about how certain quantities evolve over time, such as that mark bits once set to true remain true. The invariants 2, 3, and 4 presented in Section 3 are part of this predicate. These invariants form the crux of the hindsight reasoning, as they enable temporal interpolation.

Before we go into the formal proof argument for `delete`, we must discuss how to reason about the node-level helper functions. Figure 10 shows the specification for the helper functions assumed by the skiplist template. The specifications are logically atomic, i.e., they behave like a single atomic step in the template. The preconditions for all of the functions rely solely on the predicate `Node`. The functions `getKey`, `getHeight` and `findNext` read various components of the node. Note that `findNext` reads both the mark bit and the next pointer together.

The specification for functions `markNode` and `changeNext` is slightly more complex because they potentially change the structure. Let us explain them briefly. For `markNode` on node n at level i , the return value (`Success` or `Failure`) is determined by whether n is already marked at i . If it is, then the function returns `Failure` without modifying the node. If it is unmarked, then `markNode` successfully marks it, and updates the node accordingly. The specification for `changeNext` can be interpreted similarly. Here, the return value hinges upon the mark bit being false and the next pointer of n pointing to n' at i .

5.2 Proof of `delete`

We now have all the ingredients to show that `delete` satisfies `(HindSpec)`. We provide only a high-level summary of the proof here. Please see [36] for more details.

The precondition provides access to the invariant $\text{Inv}(r)$ and knowledge that the thread ID is tid with start time t_0 . Additionally, the thread has the right to resolve prophecy p around the decisive operations, and if the thread observes a successful decisive operation, then the atomic update $\text{AU}(\Phi)$ is available to help with the linearization. The `delete` operation begins with `traverse`. Using the \diamond operator defined in Section 4.2, we express the postcondition of `traverse` as

$$\Diamond_{s,t_0}(k \in \text{keyset}(s, c) \wedge (\text{res} \Leftrightarrow k \in C(s, c))).$$

Intuitively, this assertion captures that there is a past state s in the history (after time point t_0) in which k is in the keyset of c and res is true iff k is in the logical contents of c .

The argument here proceeds by case analysis on res . Let us first consider the case that res is *false*. The `delete` operation also terminates with *false*. Since the thread terminates without any calls to the decisive operations, this case corresponds to the $\neg\text{Upd}(pvs)$ case in the postcondition of (HindSpec). The postcondition requires `delete` to establish the predicate `PastLin(del, k, false, t0)`. In this context, establishing this predicate amounts to identifying a witness past state in which k was not part of the abstract state. Clearly, this is witnessed by state s from the specification of `traverse`. Applying (KeysetPr) in state s , we can establish the predicate `PastLin(del, k, false, t0)`.

Now, let us consider the case that res is *true*. The `maintainanceOp_del` marks node c at the higher level, but the interesting part of the proof is when the decisive operation `markNode` is called at the base level (Line 15). Again there are two cases to consider, depending on whether `markNode` succeeds. If `markNode` succeeds, then we can establish `Upd(pvs)` as we see a `Success` value being resolved. In this case, the precondition of (HindSpec) provides the atomic update $\text{AU}(\Phi)$. Since, the thread has modified the abstract state, this becomes the linearization point. The thread can linearize with $\text{AU}(\Phi)$ to obtain the receipt Φ and satisfy its postcondition. The proof also has to update the history with the new snapshot of the structure, as c goes from being unmarked to marked.

The final (and most interesting) case is when `markNode` fails. Here again, we must establish `PastLin(del, k, false, t0)` to complete the proof of (HindSpec). Two facts are useful: (i) in the past state s referred to in the `traverse` spec, we can establish that $\text{mark}(s, c) = \text{false}$; and (ii) since the `markNode` has failed, in the current state say s_0 , $\text{mark}(s_0, c) = \text{true}$. Hence, by using the second conjunct of `transition_inv` in Figure 9 and temporal interpolation on the two facts above, we can infer the existence of two consecutive states s_1 and s_2 , such that $\text{mark}(s_1, c) = \text{false}$ and $\text{mark}(s_2, c) = \text{true}$. Clearly, a concurrent `delete` thread marked c in state s_2 . Hence, this state becomes the witness to establish the predicate `PastLin(del, k, false, t0)`. This completes the proof that `delete` satisfies (HindSpec).

6 Proof Mechanization and Evaluation

We now shed light on the mechanization of the hindsight methodology, as well as its application to the skip list template. We additionally reverify the multicopy template from [34] using our new hindsight specification to modularize the proof effort. Although the multicopy algorithms are lock-based, hindsight reasoning is helpful in their verification. The case study demonstrates a substantial reduction in proof size due to the encoding of hindsight reasoning in Iris, illustrating the generality of our contribution. Our development is available as a VM and docker image on Zenodo [37].

All of the proofs we discuss below are mechanized in Iris/Coq. The templates, traversals and the node implementations are written in Iris's default programming language HeaPLang. In order to correctly capture the dependence between different layers of the proofs (such as hindsight specification and the templates, the templates and the `traverse`/node implementations), we heavily make use of Coq's module system.

The organization of our proofs is shown in Figure 11. Going from left to right, the first column relates to the formalization of hindsight reasoning in Iris. The box "Hindsight" captures the assumptions regarding the hindsight specification from Section 4. These

30:22 Verifying Lock-Free Search Structure Templates

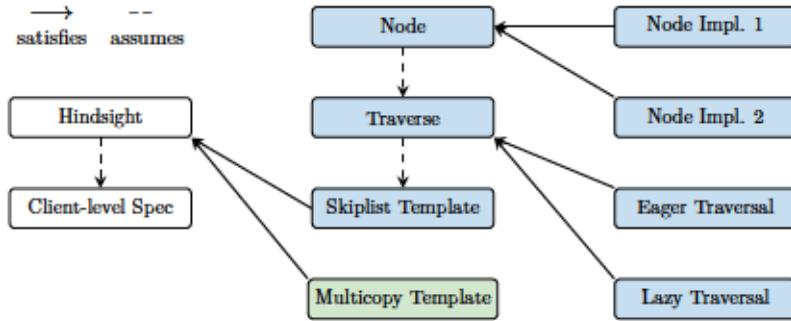


Figure 11 The structure of our proofs. Each box represents a collection of modules relevant to the label. The dashed arrows represent module dependence, i.e., assumption of specifications. The normal arrows represent implementation of the target module (fulfillment of the assumptions).

assumptions not only include the hindsight specification itself but also the relevant definitions of snapshots, histories, etc. The module “Client-level Spec” relates the client-level specification expressed in terms of atomic triples to the hindsight specification used for the template-level proofs. The corresponding proof involves the reasoning about prophecies and the helping protocol, which is done once and for all and applicable to all data structures that fulfill the assumptions made in the “Hindsight” module.

The middle column consists of modules for the two verified templates (skiplist and multicopy) and the associated proofs verifying the template operations against the hindsight specification. We discuss them in turn.

Skiplist template case study. The skiplist template, as described in Figure 2, abstracts from the concrete implementations of nodes and the `traverse` operation. Hence, we package their specifications into separate modules. To ensure that the specified data structure invariant for the skiplist template is not vacuous, we also verified an `init` routine that initializes the data structure and establishes the invariant.

The final column shows modules for the two node implementations of the skiplist template, as well as the eager and lazy traversal discussed in Section 2. The helper functions `markNode` and `changeNext` are implemented using an atomic CAS operation in both of the node implementations. The crux of the node implementation for the skiplist template is to determine a memory representation of the mark bit and the next pointer (at some level) such that both values can be read or written together with one atomic CAS operation. The first node implementation does this by using a sum type. The second node implementation is conceptually similar but uses more low-level data types instead of a sum type.

The traversal and node implementations above correspond to several existing lock-free (skip)list algorithms from the literature. The Herlihy-Shavit skiplist algorithm [10, § 14] is obtained by instantiating our template with the eager traversal, the node implementation 2, and maintenance operations that link higher-level nodes in increasing order of level and unlink nodes in the opposite order. The Michael set [31] is obtained as a degenerate case of the Herlihy-Shavit template instantiation where the skiplist is restricted to $L = 2$ (For $L = 2$, Level 1 consists of only a fixed single edge between the sentinel nodes. So, conceptually, Level 1 can be ignored in this case.)

We obtain a novel variant of a skiplist by replacing the eager traversal in the Herlihy-Shavit instantiation with the lazy traversal. The lazy traversal is inspired by the Harris list algorithm [9], which is obtained as a degenerate case of this new lazy skiplist algorithm by restricting it to $L = 2$.

Table 1 Summary of the proof effort. For each module, we show the number of lines of program code, lines of proof, total number of lines, and the proof-checking time in seconds. The code for the initialization and the core operations of the skiplist (entries with (*)) is technically defined in the “Skiplist” module, however here we present them separately for each operation to provide a better picture. The count for Herlihy-Shavit is the summation of rows “Hindsight”, “Client-level Spec”, all “Skiplist” modules, “Node Impl. 2” and “Eager Traversal”.

Skiplist Template (Iris/Coq)				
Module	Code	Proof	Total	Time
Flow Library	0	5330	5330	33
Hindsight	0	950	950	11
Client-level Spec	9	329	338	18
Skiplist	12	1693	1705	26
Skiplist Init(*)	6	319	325	15
Skiplist Search(*)	7	62	69	6
Skiplist Insert(*)	37	3457	3494	111
Skiplist Delete(*)	28	2401	2429	72
Node Impl. 1	118	908	1026	35
Node Impl. 2	106	836	942	35
Eager Traversal	38	1165	1203	96
Lazy Traversal	47	2063	2110	145
Total	408	19513	19921	603
Herlihy-Shavit	243	11212	11455	390

We present a summary of the proof effort for the skiplist template in Table 1. The proof-checking time was measured on the Docker image running on an Apple M1 Pro chip with 16GB RAM. The flow library contains the Iris formalization of the Flow Framework developed in [18, 34]. As a minor contribution, we extend this library with general lemmas for reasoning about graph updates that have an affect on an unbounded number of nodes. These lemmas are useful for the proofs of `insert`, `delete` and lazy `traverse`. The unbounded updates, as well as the maintenance operations, are the reason for the relatively high number of proof lines for the `insert` and `delete` operations.

Multicopy template case study. The multicopy template from [34] generalizes search structures such as the lock-based Log-Structured Merge (LSM) tree used widely in modern database systems. It satisfies the Map ADT specification, with `search` and `upsert` (for insert/update) as its core operations. To deal with the complexity of future-dependent external linearization points, the original proof relies on an intermediate template-level specification based on the concept of *search recency*.

Table 2 presents a detailed comparison of the multicopy template proofs from [34] versus the new proof based on the hindsight framework. The original proof consists of a total of 2779 lines. By contrast, the definitions (“Defs”) and “Client-level Spec” proofs can be factored out of the total cost of the hindsight-based proof, because it is part of the hindsight library itself. Hence, the new proof based on hindsight reasoning consists of only 1310 lines, which is a reduction of 53%. To summarize, the improvement stems from the fact that the original proof relies on an intermediate specification and a helping protocol that is tailored to multicopy structures, while our new proof uses a helping protocol that is shared among all proofs that build on the new hindsight proof method.

Table 2 Comparison of multicopy template proofs. The column “Original” shows the number of lines from the proofs in [34], while “Hindsight” shows them for our new proof effort. Module “Defs” represents definitions required for proving the client-level specification (helping invariant, history predicate, etc). Module “Client-level Spec” contains the proof relating the intermediate specification (Search Recency Specification from [34] and Hindsight Specification in our paper) to the client specification. Module “LSM” contains definitions required to instantiate the frameworks for LSM trees. Modules “Search” and “Upsert” refer to the proofs for the search and upsert operations, respectively. Entries in “()” for the “Hindsight” column are not included in the total due to being part of the hindsight library.

Multicopy Template (Iris/Coq)		
Module	Original	Hindsight
Defs	866	(950)
Client-level Spec	434	(338)
LSM	741	540
Search	411	399
Upsert	327	371
Total	2779	1310

While the majority of the reduction in the proof size stems from the elimination of structure-specific specifications and helping protocol proofs, we also saw a minor reduction in the size of the remainder of the proof. One outlier is the proof of `upsert`. Here, the increase is attributed to the fact that the proof has to construct a fresh snapshot when the operation succeeds. However, this construction is conceptually simple and could be factored out into more abstract lemmas that are provided directly by the hindsight library.

7 Related Work

The formal verification of linearizability has received much attention in recent years. We refer to [5] for a survey of relevant techniques and focus our discussion to the most closely related works.

Our work builds on the idea of template algorithms for lock-based concurrent search structures of [19, 34, 18], which we extend to the setting of lock-free implementations. A common challenge when verifying linearizability of lock-free data structures is the prevalence of future-dependent and external linearization points. Hindsight theory [32, 22, 6, 7, 26, 27] has emerged as a suitable technique to address this challenge in the context of concurrent search structures. To our knowledge, we are the first to formalize hindsight reasoning within a foundational program logic. Tools like `Poling` [40], `plankton` [26, 27], and `nekton` [25] automate hindsight reasoning at the expense of an increased trusted code base. However, these tools currently cannot handle complex data structures with unbounded outdegree like skip lists. Also, they do not aim to characterize the design space of related concurrent data structures like our template algorithms do.

Other techniques for dealing with future-dependent linearization points include arguments based on forward simulation (e.g., by tracking all possible linearizations of ongoing operations [12], tracking a partial order [17], or using commit points [3]) and backward simulation (e.g., using prophecy variables [1, 23, 15]). Our encoding of hindsight reasoning in Iris combines forward reasoning (by tracking the history of the data structure state) and backward reasoning (by using prophecies). However, the details of this encoding are for the most part hidden from the proof engineer by providing a higher-level reasoning interface

based on past predicates and temporal interpolation as proposed in [27]. Our comparison with a prior proof of multicopy structure templates [34] suggests that this abstraction helps to reduce the proof complexity.

Several works propose techniques for automatically verifying concurrent skiplists. Abdulla et al. [2] propose a technique for verifying linearizability of lock-free list-based data structures using forest automata. The evaluation considers bounded skiplists with up to 3 levels. However, the implementation does not scale to larger bounds and the unbounded case is outside the scope of the technique. We note that the height of the skiplist is tied to the expected runtime of the skiplist operations. To guarantee the expected worst-case runtime bounds, the skiplist’s height must be of order $O(\log(n))$ where n is the expected maximal number of entries in the list. For this reason, real-world skiplist implementations are also parametric in the height. Heights up to 63 levels are feasible in deployed skiplists [24], so the restriction to height 3 in [2] is unrealistic. By contrast, our proofs cover skiplists of arbitrary height.

Sánchez and Sánchez [38] present an SMT-based approach towards an automated verification of concurrent lock-based skiplists. The approach is based on a decidable theory of unbounded skiplists. However, it does not consider lock-free implementations and focuses on establishing *shape invariants* preserved by the structure instead of proving linearizability.

Unlike these automated tools, our approach does not rely on data-structure specific decidable theories for reasoning about inductive properties of heap graphs. Instead, we build on the Flow Framework [20, 21, 28], which enables local reasoning about such properties over general graphs in separation logic. As a minor contribution, we extend the mechanization of the Flow Framework from [19] with lemmas to reason about graph updates that affect properties of an unbounded number of nodes.

There are some skiplist algorithms that are not immediately covered by our template algorithm. For example, skiplists based on the algorithm presented in [8] such as Java’s `ConcurrentSkipListMap` [33] use *backlinks* to avoid restarts when a traversal fails. However, we believe that our template algorithm can be extended to subsume such algorithms by abstracting from the restart policy, similarly to how the present template abstracts from the maintenance policy.

In this paper, we assume a programming language with a garbage collected semantics. The rationale for this assumption is that issues arising from manual memory reclamation can be addressed by orthogonal means. For instance, [29, 30] propose a technique that decouples the proof of data structure correctness from that of the underlying memory reclamation algorithm, allowing the correctness proof of the data structure to be carried out under the assumption of garbage collection. Recent work also showed how to carry out such modular proofs in program logics like Iris [13].

8 Conclusions and Future Work

This paper shows how to verify some of the most challenging concurrent data structure algorithms in existence. The accompanying proofs are fully mechanized in the foundational program logic Iris. The proofs are modular and cover the broader design space of the underlying algorithms by parameterizing the verification over aspects such as the low-level representation of nodes and the style of data structure maintenance.

Besides being the first work to verify unbounded lock-free skiplists, the work has developed technologies for Iris, particularly hindsight reasoning, that can be useful in many applications.

Our proofs guarantee safety but not liveness. This limitation is shared by the algorithms they verify: in any highly concurrent (minimal or no locking) setting, a thread t may never complete because of other threads that overtake it. Fortunately, this never happens in practice where threads all advance more or less at the same pace. Verifying liveness under such fairness assumptions remains an interesting direction for future work.

Another area of future work is to verify algorithms that mix locking parts with lock-free parts both for single copy and multicopy search structures. We believe that the present framework will be a good basis for that effort.

References

- 1 Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.
- 2 Parosh Aziz Abdulla, Lukás Holík, Bengt Jonsson, Ondrej Lengál, Cong Quy Trinh, and Tomáš Vojnar. Verification of heap manipulating programs with ordered data by extended forest automata. In *ATVA*, volume 8172 of *Lecture Notes in Computer Science*, pages 224–239. Springer, 2013.
- 3 Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Suha Orhun Mutluergil. Proving linearizability using forward simulations. In *CAV (2)*, volume 10427 of *Lecture Notes in Computer Science*, pages 542–563. Springer, 2017.
- 4 Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. Tada: A logic for time and data abstraction. In *ECOOP*, volume 8586 of *Lecture Notes in Computer Science*, pages 207–231. Springer, 2014.
- 5 Brijesh Dongol and John Derrick. Verifying linearisability: A comparative survey. *ACM Comput. Surv.*, 48(2):19:1–19:43, 2015.
- 6 Yotam M. Y. Feldman, Constantin Enea, Adam Morrison, Noam Rinetzky, and Sharon Shoham. Order out of chaos: Proving linearizability using local views. In *DISC*, volume 121 of *LIPICS*, pages 23:1–23:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018.
- 7 Yotam M. Y. Feldman, Artem Khuzha, Constantin Enea, Adam Morrison, Aleksandar Nanevski, Noam Rinetzky, and Sharon Shoham. Proving highly-concurrent traversals correct. *Proc. ACM Program. Lang.*, 4(OOPSLA):128:1–128:29, 2020.
- 8 Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *PODC*, pages 50–59. ACM, 2004.
- 9 Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, volume 2180 of *Lecture Notes in Computer Science*, pages 300–314. Springer, 2001.
- 10 Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.
- 11 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
- 12 Prasad Jayanti, Siddhartha Jayanti, Ugur Yavuz, and Lizzie Hernandez. A universal, sound, and complete forward reasoning technique for machine-verified proofs of linearizability. *PACMPL*, 8(POPL), January 2024. doi:10.1145/3632924.
- 13 Jaehwang Jung, Janggun Lee, Jaemin Choi, Jaewoo Kim, Sunho Park, and Jeehoon Kang. Modular verification of safe memory reclamation in concurrent separation logic. *Proc. ACM Program. Lang.*, 7(OOPSLA2):828–856, 2023.
- 14 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018. doi:10.1017/S0956796818000151.
- 15 Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. The future is ours: prophecy variables in separation logic. *Proc. ACM Program. Lang.*, 4(POPL):45:1–45:32, 2020.

- 16 Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*, pages 637–650. ACM, 2015.
- 17 Artem Khyzha, Mike Dodds, Alexey Gotsman, and Matthew J. Parkinson. Proving linearizability using partial orders. In *ESOP*, volume 10201 of *Lecture Notes in Computer Science*, pages 639–667. Springer, 2017.
- 18 Siddharth Krishna, Nisarg Patel, Dennis E. Shasha, and Thomas Wies. Verifying concurrent search structure templates. In *PLDI*, pages 181–196. ACM, 2020.
- 19 Siddharth Krishna, Nisarg Patel, Dennis E. Shasha, and Thomas Wies. *Automated Verification of Concurrent Search Structures*. Synthesis Lectures on Computer Science. Morgan & Claypool Publishers, 2021.
- 20 Siddharth Krishna, Dennis E. Shasha, and Thomas Wies. Go with the flow: compositional abstractions for concurrent data structures. *Proc. ACM Program. Lang.*, 2(POPL):37:1–37:31, 2018.
- 21 Siddharth Krishna, Alexander J. Summers, and Thomas Wies. Local reasoning for global graph properties. In *ESOP*, volume 12075 of *Lecture Notes in Computer Science*, pages 308–335. Springer, 2020.
- 22 Kfir Lev-Ari, Gregory V. Chockler, and Idit Keidar. A constructive approach for proving data structures’ linearizability. In *DISC*, volume 9363 of *Lecture Notes in Computer Science*, pages 356–370. Springer, 2015.
- 23 Hongjin Liang and Xinyu Feng. Modular verification of linearizability with non-fixed linearization points. In *PLDI*, pages 459–470. ACM, 2013.
- 24 Meta. Facebook Open Source Library: ConcurrentSkipList. <https://github.com/facebook/folly/blob/main/folly/ConcurrentSkipList.h>. Last accessed: Apr 2024.
- 25 Roland Meyer, Anton Opaterny, Thomas Wies, and Sebastian Wolff. nekton: A linearizability proof checker. In *CAV (1)*, volume 13964 of *Lecture Notes in Computer Science*, pages 170–183. Springer, 2023.
- 26 Roland Meyer, Thomas Wies, and Sebastian Wolff. A concurrent program logic with a future and history. *Proc. ACM Program. Lang.*, 6(OOPSLA2):1378–1407, 2022.
- 27 Roland Meyer, Thomas Wies, and Sebastian Wolff. Embedding hindsight reasoning in separation logic. *Proc. ACM Program. Lang.*, 7(PLDI):1848–1871, 2023.
- 28 Roland Meyer, Thomas Wies, and Sebastian Wolff. Make flows small again: Revisiting the flow framework. In *TACAS (1)*, volume 13993 of *Lecture Notes in Computer Science*, pages 628–646. Springer, 2023.
- 29 Roland Meyer and Sebastian Wolff. Decoupling lock-free data structures from memory reclamation for static analysis. *Proc. ACM Program. Lang.*, 3(POPL):58:1–58:31, 2019.
- 30 Roland Meyer and Sebastian Wolff. Pointer life cycle types for lock-free data structures with memory reclamation. *Proc. ACM Program. Lang.*, 4(POPL):68:1–68:36, 2020.
- 31 Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*, pages 73–82. ACM, 2002.
- 32 Peter W. O’Hearn, Noam Rinetzky, Martin T. Vechev, Eran Yahav, and Greta Yorsh. Verifying linearizability with hindsight. In *PODC*, pages 85–94. ACM, 2010.
- 33 Oracle. Java concurrent skip list set. <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/concurrent/ConcurrentSkipListSet.html>. Last accessed: Jan 2024.
- 34 Nisarg Patel, Siddharth Krishna, Dennis E. Shasha, and Thomas Wies. Verifying concurrent multicopy search structures. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–32, 2021.
- 35 Nisarg Patel, Dennis Shasha, and Thomas Wies. Verifying Lock-free Search Structure Templates / Artifact. Software (visited on 2024-08-23). URL: <https://doi.org/10.4230/DARTS.10.2.15>.
- 36 Nisarg Patel, Dennis Shasha, and Thomas Wies. Verifying lock-free search structure templates. *CoRR*, abs/2405.13271, 2024. [arXiv:2405.13271](https://arxiv.org/abs/2405.13271).

30:28 Verifying Lock-Free Search Structure Templates

- 37 Nisarg Patel, Dennis Shasha, and Thomas Wies. *Verifying Lock-free Search Structure Templates / Artifact*, April 2024. Software (visited on 2024-08-23). doi:10.5281/zenodo.11051385.
- 38 Alejandro Sánchez and César Sánchez. Formal verification of skip lists with arbitrary many levels. In *ATVA*, volume 8837 of *Lecture Notes in Computer Science*, pages 314–329. Springer, 2014.
- 39 Dennis E. Shasha and Nathan Goodman. Concurrent search structure algorithms. *ACM Trans. Database Syst.*, 13(1):53–90, 1988.
- 40 He Zhu, Gustavo Petri, and Suresh Jagannathan. Poling: SMT aided linearizability proofs. In *CAV (2)*, volume 9207 of *Lecture Notes in Computer Science*, pages 3–19. Springer, 2015.

Ozone: Fully Out-of-Order Choreographies

Dan Plyukhin 

University of Southern Denmark, Odense, Denmark

Marco Peressotti 

University of Southern Denmark, Odense, Denmark

Fabrizio Montesi 

University of Southern Denmark, Odense, Denmark

Abstract

Choreographic programming is a paradigm for writing distributed applications. It allows programmers to write a single program, called a choreography, that can be compiled to generate correct implementations of each process in the application. Although choreographies provide good static guarantees, they can exhibit high latency when messages or processes are delayed. This is because processes in a choreography typically execute in a fixed, deterministic order, and cannot adapt to the order that messages arrive at runtime. In non-choreographic code, programmers can address this problem by allowing processes to execute out of order – for instance by using futures or reactive programming. However, in choreographic code, out-of-order process execution can lead to serious and subtle bugs, called *communication integrity violations (CIVs)*.

In this paper, we develop a model of choreographic programming for out-of-order processes that guarantees absence of CIVs and deadlocks. As an application of our approach, we also introduce an API for safe non-blocking communication via futures in the choreographic programming language Choral. The API allows processes to execute out of order, participate in multiple choreographies concurrently, and to handle unordered data messages. We provide an illustrative evaluation of our API, showing that out-of-order execution can reduce latency and increase throughput by overlapping communication with computation.

2012 ACM Subject Classification Computing methodologies → Concurrent computing methodologies

Keywords and phrases Choreographic programming, Asynchrony, Concurrency

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.31

Related Version *Full Version:* <https://arxiv.org/abs/2401.17403> [27]

Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact):* <https://doi.org/10.4230/DARTS.10.2.16>
Software (Source Code): <https://github.com/dplyukhin/ozone>

Funding Partially supported by Villum Fonden (grant no. 29518). Co-funded by the European Union (ERC, CHORDS, 101124225). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

1 Introduction

Choreographic programming [25] is a paradigm that simplifies writing distributed applications. In contrast to a traditional development style, where one implements a separate program for each type of process in the system, choreographic programming allows a programmer to define the behaviors of all processes together in a single program called a *choreography* [26]. Through *endpoint projection (EPP)*, a choreography can be compiled to generate the programs implementing each process that would otherwise need to be written by hand. Aside from convenience, the advantage of this approach is that certain classes of bugs (such as deadlocks)

 © Dan Plyukhin, Marco Peressotti, and Fabrizio Montesi;
licensed under Creative Commons License CC-BY 4.0
38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 31; pp. 31:1–31:28



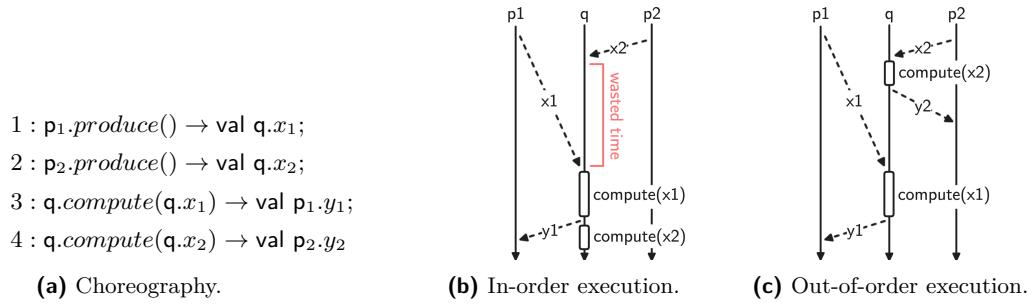


Figure 1 A choreography where out-of-order execution can improve performance.

are impossible *by construction* [8]. Choreographic programming has been applied to popular languages such as Java [16] and Haskell [32], and has been used to implement real-world protocols such as IRC [23].

Processes in choreographic programs typically execute in a fixed, sequential order. Consider Figure 1a, which shows a simple choreography performed by processes p_1 , p_2 , and q . The syntax $p.e \rightarrow \text{val } q.x$ means “ p evaluates expression e and sends the result to q , which binds the result to a local variable x ”. Under the usual semantics for choreographies, $p_1.produce()$ and $p_2.produce()$ can be evaluated in parallel because p_1 and p_2 are distinct processes. However, q must execute each step sequentially: first q waits until it receives x_1 ; then q waits until it receives x_2 ; and only then can q send p_1 the result of processing x_1 .

Figure 1b depicts an execution of the choreography, showing the drawback of a fixed processing order: if x_2 arrives before x_1 , q wastes time waiting for x_1 instead of processing x_2 . Ideally, q would evaluate $compute(q.x_1)$ and $compute(q.x_2)$ according to the arrival order of x_1 and x_2 , as shown in Figure 1c. Assuming these two expressions are safe to reorder, such an optimization would allow q to overlap computation with communication and reduce the average latency experienced by p_1 and p_2 . We are therefore interested in studying choreographic programming models where processes may execute some statements out of order, or even concurrently. We call such processes *out-of-order processes* and the corresponding choreographies (*fully*) *out-of-order choreographies*.

Processes with out-of-order features have been considered in prior work. Process models such as the actor model [2] or the π -calculus with delayed receive [24] are expressive enough to implement the behavior in Figure 1c, but these models lack the static guarantees of choreographic programming. More recently, Montesi gave a semantics for *nondeterministic choreographies* [26], i.e., choreographies with nondeterministic choice. Nondeterministic choreographies can implement the execution in Figure 1c, but they are unwieldy when it comes to expressing out-of-order process execution: they require explicitly writing all possible schedulings, lest getting a suboptimal program. For our example, we would get a choreography twice the size of the one in Figure 1a (cf. Section 6). Consequently, nondeterministic choreographies are both hard to write and brittle – a typical drawback when using syntactic operators to express interleavings. This raises the question:

Can we develop a choreographic programming model for out-of-order processes that marries the simple syntax of Figure 1a with the semantics of Figure 1c?

The simplicity of this problem is deceptive, since common-sense approaches can lead to pernicious compiler bugs. For instance, consider Figure 2: two microservices cs (a “content service”) and ks (a “key service”) send values txt, key to a server s (lines 1 and 2). The server

in turn forwards those values to a client c (lines 3 and 4). Notice that if s is an out-of-order process, then it can forward the results in any order, as shown in Figures 2b and 2c. This causes a problem for c : since both txt and key were sent by s , and since both values have the same type (`String`), c has no way to determine whether the first message contains txt (as in Figure 2b) or key (as in Figure 2c). This problem is easy for compiler writers to miss, leading to disastrous nondeterministic bugs where variables are bound to the wrong values. We call such bugs *communication integrity violations (CIVs)*.

In this paper, we investigate CIVs and other complications that arise from mixing choreographies with out-of-order processes. Our investigation brings forward necessary elements that are missing from previous research on choreographic programming [26] and the neighbouring approach of multiparty session types (which use simpler choreographies without data or computation) [19, 15, 1, 33]. Although the problem in Figure 2 can easily be solved by attaching static information (such as variable names) to each message, we show in Section 2 that a general solution requires mixing static and dynamic information, replicated across multiple processes. We also find that formalizing fully out-of-order choreographies requires several features uncommon in standard choreographic programming models, such as scoped variables and an expanded notion of well-formedness.

We make the following key contributions:

1. We present O_3 , a formal model for asynchronous, fully out-of-order choreographies.¹ Our model prevents CIVs by attaching *integrity keys* to messages. A nice consequence of our solution is that messages no longer need to be delivered in FIFO order. We prove that O_3 choreographies ensure deadlock-freedom (Theorem 2) and communication integrity (Theorem 4).
2. We present an EPP algorithm to project O_3 choreographies into out-of-order processes. We prove an operational correspondence theorem, which states that a choreography and its projection evolve in lock-step (Theorem 6). The key to making this proof tractable is a new notion of well-formedness that formalizes a communication integrity invariant. The theorem implies that a correct compiler will not generate code with deadlocks or CIVs.
3. We demonstrate the applicability of our approach by developing *Ozone*, a non-blocking communication API for the choreographic programming language Choral [16].² Choreographies implemented with Ozone can use futures [4] to process messages concurrently (as in Figure 1c) without violating communication integrity. We evaluate Ozone with microbenchmarks and a model serving pipeline [34]. Our results confirm that out-of-order execution can dramatically reduce latency and increase throughput for choreographies, putting the performance of hand-written reactive processes within reach of choreographic programmers (we compare to actors written in the popular *Akka* framework [3]).

The paper is structured as follows. Section 2 explores CIVs and other issues in out-of-order choreography models. Section 3 presents our formal model O_3 . Section 4 presents our model for out-of-order processes and our EPP. Section 5 presents our non-blocking API for Choral and our evaluation. We conclude with related work in Section 6 and discussion in Section 7.

2 Overview

In this section we explore the challenges that must be solved to develop a fully out-of-order choreography model, along with our approach.

¹ The name O_3 derives from our model being *Out Of Order*.

² The name *Ozone* derives from O_3 being the chemical formula for ozone.

31:4 Ozone: Fully Out-of-Order Choreographies

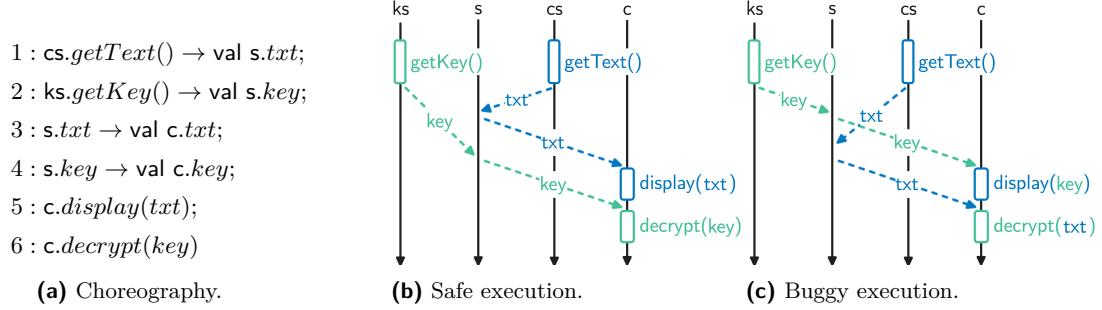


Figure 2 A choreography where naïve out-of-order execution is unsafe. Process **c** cannot distinguish whether the first message it receives represents *key* or *txt*.

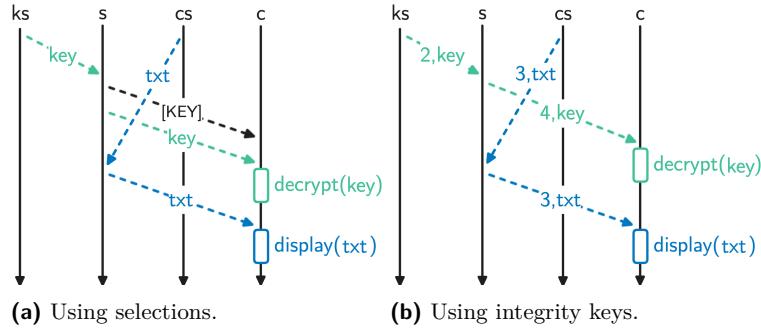


Figure 3 Two approaches to prevent CIVs: selections and integrity keys.

2.1 Intraprocedural Integrity

Informally, communication integrity is the property that messages communicated in a choreography are bound to the correct variables. To ensure this, processes might need extra information; in Figure 2, process **c** needs to know which value will arrive first: *txt* or *key*.

A traditional solution would be for **s** to send a *selection* to **c**. Selections are communications of constant values, used in choreography languages when one process makes a control flow decision that other processes must follow. Figure 3a shows how **s** could send the selection [KEY] to inform **c** that *key* will arrive before *txt*. Indeed, this is the approach used by nondeterministic choreographies [26]. However, selections impose overhead: any time nondeterminism could occur, the programmer would need to insert new selection messages. These extra messages would have both a cognitive cost for the programmer (as programs become littered with selections) and a runtime cost in the form of an extra message.

Instead, we opt to pair each message with a disambiguating tag called an *integrity key*. When **c** receives a message, it checks the integrity key to find the meaning of the message. Figure 3b uses *line numbers* as integrity keys. For example, the *txt* message is tagged with the number 3 because it was produced by the instruction on line 3 in Figure 2. Equivalently, one could use variable names (assuming that all variables have distinct names), message types (assuming that all messages have distinct types), or operators [7]; essentially these are all ways to combine messages with selections. However, as we will see in the next section, none of these solutions will suffice once we introduce procedures and recursion.

Integrity keys have another advantage over selections: they make it safe for the network to reorder messages. For instance, the selection in Figure 3a will only prevent CIVs if *key* and *txt* are delivered in the same order they were sent. Thus previous theories and implementations

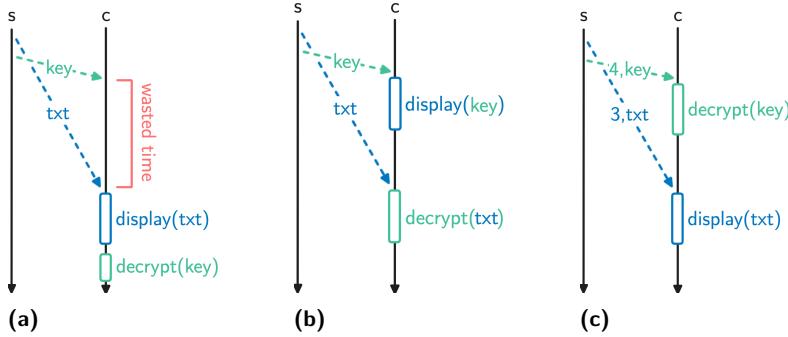


Figure 4 The challenges of non-FIFO delivery. Part (a) depicts head-of-line blocking when using a FIFO transport protocol: The message containing k arrives first, but it cannot be processed until t arrives. Part (b) depicts a CIV caused by using an unordered transport protocol without integrity keys. Part (c) depicts how the processes can use integrity keys to prevent CIVs.

of choreographic languages require a transport protocol that ensures reliable FIFO communication [16, 26]. These models are therefore susceptible to head-of-line blocking [31], where one delayed message can prevent others from being processed (Figure 4a). Figure 4b shows why FIFO is necessary in these models: unordered messages can cause CIVs. Because our model combines unordered messages, integrity keys, and out-of-order processes, it circumvents the head-of-line blocking problem – as shown in Figure 4c.

2.2 Procedural Choreographies

Choreographies can use procedures parameterised on processes for modularity and recursion [11, 26]. Figure 5a shows an example: a procedure X with three *roles* (i.e., process parameters) a, b, c . The procedure X is invoked twice – once with processes p, q, r_1 (line 7) and again with p, q, r_2 (line 8). In the body of X , role a produces a value and sends it to b ; then b transforms the value and sends it to c ; finally, c processes the value and sends it to a . As usual in most programming languages, we will assume the variables $a.w, b.x, c.y$, and $a.z$ are locally scoped – this is in contrast to many choreography models, where variables at processes are all mutable fields accessible anywhere in the program.

In existing choreography models, a process can only participate in one choreographic procedure at a time. This is no longer the case with fully out-of-order choreographies. Consider Figure 5a, where process p invokes procedure X twice. The process may begin by invoking the first procedure call (line 7), computing $p.w$ (line 2), and sending $p.w$ to r_1 (line 3). Then, instead of executing its next instruction – i.e. becoming blocked by waiting for a message on line 5 – p can skip the instruction and proceed to invoke the second procedure call (line 8). Thus, we can have an execution like in Figure 5b, in which p sends a message to r_1 as part of the first procedure call and immediately sends a message to r_2 as part of the second procedure call. This unusual semantics is exactly what we would expect in a choreography language with non-blocking receive – such as in Choral, when using the Ozone API to bind the result of a communication to a future (Section 5).

2.2.1 Interprocedural Integrity

Concurrent choreographic procedures add another dimension of complexity to the communication integrity problem. Figures 5b and 5c show why: depending on the order that r_1 and r_2 's messages arrive at q , the messages from q may arrive at p in any order. (This occurs even

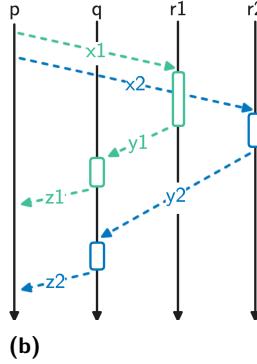
31:6 Ozone: Fully Out-of-Order Choreographies

```

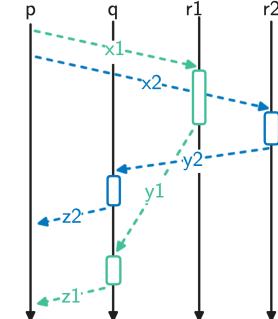
1 : X(a, b, c) =
2 :   val a.w = produce();
3 :   a.w → val b.x;
4 :   b.transform(x) → val c.y;
5 :   c.process(y) → val a.z;
6 :   a.store(w, z)
7 : X(p, r1, q);
8 : X(p, r2, q)

```

(a)



(b)



(c)

Figure 5 A choreography and two possible executions. In both diagrams, the green lines correspond to $X(p, q, r_1)$ and the blue lines correspond to $X(p, q, r_2)$.

if we assume reliable FIFO delivery!) Like in the previous section, p cannot distinguish which message pertains to which procedure invocation. But now static information is insufficient to ensure communication integrity: both messages from q pertain to the same variable in the same procedure, so the integrity keys fail to distinguish the different procedure calls. We call this the *interprocedural CIV problem*.

The example above shows that integrity keys need dynamic information prevent CIVs. We can solve the problem by combining the line numbers used in Section 2.1 with some *session token* t that uniquely identifies each procedure invocation. Applied to Figures 5b and 5c, p could inspect the session token to determine whether the messages pertain to the first procedure call (line 7) or the second (line 8). But this requires p and q to somehow achieve *a priori* agreement about which tokens correspond to which procedure invocations.

One solution to the interprocedural CIV problem would be to select a “leader” process for each procedure call, and let the leader compute a session token for all the other roles to use. However, this would make the leader a bottleneck: until the other participants receive the token, senders would not be able to send messages, and recipients would not be able to discern which procedure invocation their incoming messages pertain to. We therefore propose a method for processes to compute session tokens independently, using only local data, such that they still agree on the same value of the token for each procedure invocation.

Observe that a procedure call is uniquely identified by its caller (i.e. the procedure call that called it) and its line number l. Assuming the caller already has a unique token t, the callee’s token can be computed as some injective function $\text{nextToken}(l, t)$. This function would need to satisfy two properties:

- **Determinism:** For any input pair l, t , $\text{nextToken}(l, t)$ always produces the same value t' .
- **Injectivity:** Distinct input pairs l, t produce distinct output tokens.

Determinism ensures that if two processes in the same procedure call (with token t) invoke the same procedure (on line l) then both processes will agree on the value of $\text{nextToken}(l, t)$. *Injectivity* ensures that if a process concurrently participates in two different procedure calls (with distinct tokens t_1, t_2) and invokes two procedures (on lines l_1, l_2 – possibly $l_1 = l_2$) then the resulting session tokens will be distinct ($\text{nextToken}(l_1, t_1) \neq \text{nextToken}(l_2, t_2)$). In the next section, we realize these constraints by representing tokens as lists of line numbers and defining nextToken to be the list-prepend operator.

$\mathcal{C} ::= \{X_i(\bar{p}, \bar{p}.x) = C_i\}_{i \in \mathcal{I}}$	<i>(decls)</i>
$C ::= I; C$	<i>(seq)</i>
$ 0$	<i>(end)</i>
$I ::= l, t : p.e \rightarrow \text{val } q.x$	<i>(comm)</i>
$ l, t : \text{val } p.x = e$	<i>(expr)</i>
$ l, t : X(\bar{p}, \bar{a})$	<i>(call)</i>
$ l, t : p \rightsquigarrow q[L]$	<i>(sel[†])</i>
$t ::= t$	<i>(placeholder)</i>
$e ::= f(\bar{e})$	<i>(app)</i>
$a ::= v@p$	<i>(val)</i>
	<i>(p.x)</i>
	<i>(var)</i>
	<i>(block)</i>
	<i>(sel)</i>
	<i>(cond)</i>
	<i>(comm[†])</i>
	<i>(call[†])</i>
	<i>(token[†])</i>
	<i>(atom)</i>

Figure 6 Syntax for choreographies in O_3 . Terms marked with \dagger only appear at runtime.

3 Choreography Model

In this section we present O_3 , a formal model for asynchronous, fully out-of-order choreographies. Statements can be executed in any order (up to data dependency) and messages can be delivered out of order. The section concludes with proofs of deadlock-freedom and communication integrity.

3.1 Syntax

The syntax for choreographies in O_3 is defined by the grammar in Figure 6. Two example choreographies are shown in Figure 7; we explain their semantics in Section 3.2.2.

A choreography C is executed in the context of a collection of *procedures* \mathcal{C} . Each procedure $X_i(\bar{p}, \bar{p}.x) = C_i$ is parameterized by a list of *roles* $\bar{p} = p_1, \dots, p_n$ and role-local *parameters* $\bar{p}.x = p_{j_1}.x_1, \dots, p_{j_m}.x_m$ where every parameter $p_{j_k}.x_k$ is located at one of the roles in \bar{p} . We assume that procedures do not contain runtime terms (such as $l, t : p \rightsquigarrow q.x$).

A choreography C consists of a sequence of *instructions* I , followed by the end symbol 0 which is often omitted. Each instruction is prefixed with a *line number* l and a *token* t . We call this pair an *integrity key*. If C_i is the body of a procedure in \mathcal{C} , then the token t on every instruction in C_i must be a *token placeholder* t . When the procedure is invoked, all token placeholders t in C_i will be replaced with a fresh *token value* τ .

We assume that line numbers in \mathcal{C} are similar to line numbers in a real computer program: Each instruction I in \mathcal{C} has a distinct line number l . When a procedure $X_i(\bar{p}, \bar{p}.x) = C_i$ is invoked, the line numbers in C_i will remain unchanged. This will allow us to access the static location of an instruction at runtime in order to compute the integrity key.

There are five kinds of instructions. A communication $p.e \rightarrow \text{val } q.x; C$ instructs process p to evaluate expression e and send it to process q , which will bind the result to $q.x$ in the continuation C . A selection $p \rightarrow q[L]; C$ conveys knowledge of choice [26]: it instructs p to send a value literal L to q , informing q that a decision (represented by L) has been made. A local computation $\text{val } p.x = e; C$ instructs p to evaluate e and bind the result to $p.x$ in C . A conditional $\text{if } e@p \text{ then } C_1 \text{ else } C_2; C$ instructs p to evaluate e and for the processes to proceed with C_1 or C_2 according to the result. A procedure call $X_i(\bar{p}, \bar{a}); C$ instructs processes \bar{p} to invoke procedure $X_i(\bar{q}, \bar{q}.y) = C_i$ defined in \mathcal{C} , with processes \bar{p} playing roles

$BuyItem(s, b, b.itemID) =$ 1, t : b.itemID → val s.itemID; 2, t : val s.item? = sell(s.itemID); 3, t : s.item? → val b.item? 4, $\tau_0 : BuyItem(\text{seller}, \text{buyer}_1, 123@\text{buyer}_1)$; 5, $\tau_0 : BuyItem(\text{seller}, \text{buyer}_2, 543@\text{buyer}_2)$	$StreamIt(p, c) =$ 1, t : p.produce() → val c.x; 2, t : val c.z = consume(c.x); 3, t : if (itemsLeft() > 0)@p then 4, t : p → c[MORE] 5, t : StreamIt(p, c) else 6, t : p → c[DONE] 7, $\tau_0 : StreamIt(p_1, c)$; 8, $\tau_0 : StreamIt(p_2, c)$
(a)	(b)

Figure 7 Two example choreographies. On the left, processes buyer_1 and buyer_2 concurrently attempt to buy products from seller . On the right, producers p_1 and p_2 concurrently send streams of data to a shared consumer c .

\bar{q} and arguments \bar{a} (which may take the form of values $v@p$ or variables $p.x$) substituted for parameters $\bar{q}.\bar{y}$ in C_i . In addition to these basic instructions, a choreography may contain blocks $\{C\}; C'$ which limit the scope of variables defined in C so they do not extend to C' .

In addition, choreographies can contain *runtime instructions* that represent an instruction in progress; these terms are an artifact of the semantics, not written explicitly by the programmer. A communication-in-progress $p \rightsquigarrow q.x$ indicates that p sent a message to q , which q has not yet received. Similarly, a selection-in-progress $p \rightsquigarrow q[L]$ indicates that p sent a selection. A procedure-call-in-progress $\bar{p}.X(\bar{q}, \bar{a})\{C\}$ indicates that some processes have invoked X , and others have not – we leave the details to Section 3.2.

Expressions e are composed of *atoms* a (i.e. variables $p.x$ and values $v@p$) and function applications $f(\bar{e})$. Although the variables $p.x$ are immutable, we assume that a function f evaluated by p can mutate p 's state as a side-effect. Technically, having side-effects in our theory is not necessary. However, most choreographic programming theories and implementations equip processes with mutable state [26]; this includes Choral, the language we use to implement the Ozone API in Section 5.

3.2 Semantics

We now give a fully out-of-order semantics for choreographies in O_3 . The semantics is a labelled transition system on *configurations* $\langle C, \Sigma, K \rangle$, where C is a choreography, Σ is a mapping from process names p to process states σ , and K is a mapping from process names p to multisets of messages M yet to be delivered to p . We also assume there exists a set of unchanging procedure declarations \mathcal{C} , not shown explicitly in the configuration.

An *initial configuration* is a configuration $\langle C, \Sigma, K \rangle$ where Σ maps each p to an arbitrary state, K maps each p to the empty set, and all instructions in C use the same token τ_0 , called the *initial token*. We assume initial configurations to be well-formed, cf. Section 3.3. The transition relation (\xrightarrow{p}) is on configurations, where p identifies which process took a step.

Messages in our semantics are represented as triples (l, τ, v) . Here l is the line number of the communication that sent the message, τ is the token associated with the procedure invocation that sent the message, and v is a value called the *payload*. Together, the pair (l, τ) is called the *integrity key* of the message; the line number prevents intraprocedural CIVs (Section 2.1) while the token prevents interprocedural CIVs (Section 2.2).

3.2.1 Transition rules

Figure 8 defines the semantics for O_3 , which extends textbook models for procedural and asynchronous choreographies to allow full out-of-order execution [26]. That is, in a choreography of the form $I_1; I_2; C$, the statement I_2 can always be executed before I_1 unless:

1. (*Data dependency*) I_1 binds a variable $p.x$ that is used in I_2 ; or
2. (*Control dependency*) I_1 is a selection of the form $p \rightarrow q[L]$ or $p \rightsquigarrow q[L]$, and I_2 is an action performed by q .

The semantics for communication is defined by rules C-SEND and C-RECV. In C-SEND for the communication term $l, \tau : p.e \rightarrow val q.x$, the expression e is evaluated in the context of p 's state using the notation $\Sigma(p) \vdash e \Downarrow (v, \sigma)$. Evaluating e produces a value v and a new state σ for p ; we assume that (\vdash) is defined for any e that contains no free variables and for any state $\Sigma(p)$. The C-SEND rule transforms the communication term into a communication-in-progress term $l, \tau : p \rightsquigarrow q.x$ and adds the message (l, τ, v) to q 's set of undelivered messages. The message can subsequently be received by q using the C-RECV rule. This rule removes the communication-in-progress term and substitutes the message payload v into the continuation C . Notice that the integrity key l, τ of the message is matched against the integrity key of the communication-in-progress, $l, \tau : p \rightsquigarrow q.x$. Notice also that the semantics for communication is not defined if the token t is merely a placeholder t – it must be a token *value* τ . Indeed, in Section 3.3 we show that placeholders only appear in \mathcal{C} , never in C .

Rules C-SELECT and C-ONSELECT closely mirror the semantics of C-SEND and C-RECV – the key difference is that a label L is communicated instead of a value. Rules C-COMPUTE and C-IF are standard, except for changes made to use lexical scope instead of global scope: C-COMPUTE substitutes the value v into the continuation C (instead of storing it in the local state Σ) and C-IF places the continuation C_i in a block to prevent variable capture. To garbage collect empty blocks, C-IF uses a concatenation operator (\circ) defined as:

$$\{I; C\} \circ C' = \{I; C\}; C' \quad \{0\} \circ C' = C'$$

The C-DELAY rule is used in choreography models to enable a limited form of out-of-order execution, where unrelated processes execute concurrently: given a choreography $I; C$, C-DELAY would ordinarily prevent any q from executing in C if q is somehow involved in I . Our formulation of the rule is weakened: q is only prevented from executing in C if I is a selection at q , i.e. a control dependency. The rule still respects data dependencies, however, by design of the other rules – for instance, $l, \tau : p.x \rightarrow val q.y$ cannot be evaluated until x is bound to a value. Thus our version of C-DELAY enables *full* out-of-order execution.

The rules C-FIRST, C-ENTER, C-LAST, and C-DELAY-PROC model procedure calls, with extra machinery to model how processes can execute their roles in a choreographic procedure in parallel until they need to interact. Given a procedure call $l, \tau : X(\bar{p}, \bar{a})$, C-FIRST models how $p \in \bar{p}$ has entered the procedure before any of the other processes. The rule replaces the procedure call with a procedure-call-in-progress $l, \tau : \bar{p} \setminus p. X(\bar{p}, \bar{a}) \{ C'_1 \}$ to reflect this fact; the choreography C'_1 is the body of the procedure, which p may begin executing via the C-DELAY-PROC rule. The remaining processes can enter the procedure via the C-ENTER rule, and the last process to enter the procedure uses the C-LAST rule. As we explain below, these rules also compute new integrity keys for the callee procedure to prevent CIVs.

The key novelty of our semantics for procedures is the use of `nextToken`. In C-FIRST, the body C'_1 is obtained by computing the token $\tau' = \text{nextToken}(l, \tau)$ and substituting τ' for all occurrences of the token placeholder t . Notice that the semantics makes it appear as if the processes have synchronized to compute the next token; in Section 4, we give a semantics where each process computes the next token independently and in Theorem 6 we prove that the two models correspond. Hence the apparent synchronization has no runtime cost.

As discussed in Section 2.2.1, $\text{nextToken} : \mathbb{N} \times \text{Token} \rightarrow \text{Token}$ is a pure injective function for computing new tokens (of type `Token`) using integrity keys (of type $\mathbb{N} \times \text{Token}$). To ensure the integrity keys from two concurrent procedures never collide, `nextToken` must produce unique, non-repeating keys upon iterated application. One way this can be realized is by representing `Token` = \mathbb{N}^* as lists of numbers, the initial token τ_0 as an empty list $[]$, and implementing $\text{nextToken}(l, \tau) = l :: \tau$, i.e. prepending the line number l to the list. Intuitively, this means the token associated with a procedure invocation is a simplified *call stack* of line numbers from which the procedure was called.

3.2.2 Discussion

Figure 7a expresses a choreography in which two buyer processes concurrently buy items from a seller process. In the initial configuration, `buyer1` can enter the procedure on line 4, `buyer2` can enter the procedure on line 5, and `seller` can enter either procedure. If `buyer2` enters first (using C-DELAY and C-ENTER), it can proceed to send 543@`buyer2` to `seller` (using C-COM). Then `seller` can enter the procedure on line 5 (using C-DELAY and C-LAST) and proceed to receive the message from `buyer2` (using C-RECV). This execution would be impossible in a standard choreography model because `seller` would need to complete the procedure invocation on line 4 before it could enter the procedure on line 5. The added concurrency ensures that slowness in `buyer1` does not prevent `buyer2` from making progress.

Notice the out-of-order semantics of Figure 7a also adds nondeterminism. Suppose `buyer1` and `buyer2` attempt to buy the same item and the `seller` only has one copy. One of the buyers will receive the item, and the other will receive a null value. In a standard choreography model, the item would always go to `buyer1`. In O_3 , the item will be sold nondeterministically according to the order that messages arrive to the seller. This nondeterminism can be problematic – it makes reasoning about choreographies harder – but also increases expressivity: nondeterminism is essential in distributed algorithms like consensus and leader election. Reasoning about nondeterminism in choreographies is an important topic for future work.

Figure 7b shows we can also express recursive choreographies. In each iteration of the procedure *StreamIt*, a producer `p` sends a value to a consumer `c` (line 1) and decides whether to start another iteration (line 3). Then the producer asynchronously informs the consumer about its decision (lines 4 and 6) and can proceed with the next iteration (line 5) without waiting for the consumer. Because messages in O_3 are unordered, the consumer can consume items (line 2) from different iterations in any order; this prevents head-of-line blocking [31].

In the initial choreography of Figure 7b, producers `p1, p2` and a consumer `c` invoke two instances of *StreamIt*. As in Figure 7a, the two procedures evolve concurrently; a slowdown in `p1` will not prevent `c` from consuming items produced by `p2`.

3.3 Properties

In this section we prove that O_3 choreographies are deadlock-free and we formalize the communication integrity property. Combined with the EPP Theorem presented in Section 4, these results imply that projected code inherits the same properties.

$$\begin{array}{c}
\frac{\Sigma(\mathbf{p}) \vdash e \Downarrow (v, \sigma) \quad M = K(\mathbf{q}) \uplus \{(l, \tau, v)\}}{\langle l, \tau : \mathbf{p}.e \rightarrow \mathbf{val} \; \mathbf{q}.x; C, \Sigma, K \rangle \xrightarrow{\mathbf{p}} \langle l, \tau : \mathbf{p} \rightsquigarrow \mathbf{q}.x; C, \Sigma[\mathbf{p} \mapsto \sigma], K[\mathbf{q} \mapsto M] \rangle} \text{C-SEND} \\
\\
\frac{(l, \tau, v) \in K(\mathbf{q}) \quad M = K(\mathbf{q}) \setminus \{(l, \tau, v)\}}{\langle l, \tau : \mathbf{p} \rightsquigarrow \mathbf{q}.x; C, \Sigma, K \rangle \xrightarrow{\mathbf{q}} \langle C[\mathbf{q}.x \mapsto v @ q], \Sigma, K[\mathbf{q} \mapsto M] \rangle} \text{C-RECV} \\
\\
\frac{M = K(\mathbf{q}) \cup \{(l, \tau, L)\}}{\langle l, \tau : \mathbf{p} \rightarrow \mathbf{q}[L]; C, \Sigma, K \rangle \xrightarrow{\mathbf{p}} \langle l, \tau : \mathbf{p} \rightsquigarrow \mathbf{q}[L]; C, \Sigma, K[\mathbf{q} \mapsto M] \rangle} \text{C-SELECT} \\
\\
\frac{K(\mathbf{q}) = \{(l, \tau, L)\} \cup M}{\langle l, \tau : \mathbf{p} \rightsquigarrow \mathbf{q}[L]; C, \Sigma, K \rangle \xrightarrow{\mathbf{q}} \langle C, \Sigma, K[\mathbf{q} \mapsto M] \rangle} \text{C-ONSELECT} \\
\\
\frac{\Sigma(\mathbf{p}) \vdash e \Downarrow (v, \sigma)}{\langle l, \tau : \mathbf{val} \; \mathbf{p}.x = e; C, \Sigma, K \rangle \xrightarrow{\mathbf{p}} \langle C[\mathbf{p}.x \mapsto v @ \mathbf{p}], \Sigma[\mathbf{p} \mapsto \sigma], K \rangle} \text{C-COMPUTE} \\
\\
\frac{\Sigma(\mathbf{p}) \vdash e \Downarrow v \quad \text{if } v = \mathbf{true} \text{ then } i = 1 \text{ else } i = 2}{\langle l, \tau : \text{if } e @ \mathbf{p} \text{ then } C_1 \text{ else } C_2; C, \Sigma, K \rangle \xrightarrow{\mathbf{p}} \langle \{C_i\}; C, \Sigma, K \rangle} \text{C-IF} \\
\\
\frac{\langle C_1, \Sigma, K \rangle \xrightarrow{\mathbf{p}} \langle C'_1, \Sigma', K' \rangle}{\langle \{C_1\}; C_2, \Sigma, K \rangle \xrightarrow{\mathbf{p}} \langle \{C'_1\}; C_2, \Sigma', K' \rangle} \text{C-BLOCK} \\
\\
\frac{\langle C, \Sigma, K \rangle \xrightarrow{\mathbf{q}} \langle C', \Sigma', K' \rangle \quad I \text{ is not a selection at } \mathbf{q}}{\langle I; C, \Sigma, K \rangle \xrightarrow{\mathbf{q}} \langle I; C', \Sigma', K' \rangle} \text{C-DELAY} \\
\\
\frac{(X(\bar{\mathbf{q}}, \bar{\mathbf{q}}.y) = C_1) \in \mathcal{C} \quad C'_1 = C_1[\bar{\mathbf{q}}, \bar{\mathbf{q}}.y, \mathbf{t} \mapsto \bar{\mathbf{p}}, \bar{a}, \tau'] \quad \mathbf{p} \in \bar{\mathbf{p}} \quad \tau' = \text{nextToken}(l, \tau)}{\langle l, \tau : X(\bar{\mathbf{p}}, \bar{a}); C_2, \Sigma, K \rangle \xrightarrow{\mathbf{p}} \langle l, \tau : \bar{\mathbf{p}} \setminus \mathbf{p}.X(\bar{\mathbf{p}}, \bar{a}) \{C'_1\}; C_2, \Sigma, K \rangle} \text{C-FIRST} \\
\\
\frac{\mathbf{p} \in \bar{\mathbf{p}}}{\langle l, \tau : \bar{\mathbf{p}}.X(\bar{\mathbf{q}}, \bar{a}) \{C_1\}; C_2, \Sigma, K \rangle \xrightarrow{\mathbf{p}} \langle l, \tau : \bar{\mathbf{p}} \setminus \mathbf{p}.X(\bar{\mathbf{q}}, \bar{a}) \{C_1\}; C_2, \Sigma, K \rangle} \text{C-ENTER} \\
\\
\frac{\langle C_1, \Sigma, K \rangle \xrightarrow{\mathbf{p}} \langle C'_1, \Sigma', K' \rangle \quad \mathbf{p} \notin \bar{\mathbf{p}}}{\langle l, \tau : \bar{\mathbf{p}}.X(\bar{\mathbf{q}}, \bar{a}) \{C_1\}; C_2, \Sigma, K \rangle \xrightarrow{\mathbf{p}} \langle \{C_1\}; C_2, \Sigma, K \rangle} \text{C-LAST} \\
\\
\frac{\langle C_1, \Sigma, K \rangle \xrightarrow{\mathbf{p}} \langle C'_1, \Sigma', K' \rangle}{\langle l, \tau : \bar{\mathbf{p}}.X(\bar{\mathbf{q}}, \bar{a}) \{C_1\}; C_2, \Sigma, K \rangle \xrightarrow{\mathbf{p}} \langle l, \tau : \bar{\mathbf{p}}.X(\bar{\mathbf{q}}, \bar{a}) \{C'_1\}; C_2, \Sigma', K' \rangle} \text{C-DELAY-PROC}
\end{array}$$

Figure 8 Semantics for fully out-of-order choreographies.

31:12 Ozone: Fully Out-of-Order Choreographies

$$\begin{array}{c}
\frac{\forall v, (l, \tau, v) \notin K(\mathbf{q})}{\langle l, \tau : p.e \rightarrow val q.x, K \rangle \checkmark} \text{ C-WF-SEND} \quad \frac{(l, \tau, L) \notin K(\mathbf{q})}{\langle l, \tau : p \rightarrow q[L], K \rangle \checkmark} \text{ C-WF-SELECT} \\
\\
\frac{\begin{array}{c} \overline{p} \text{ distinct} \quad \overline{p.x} \text{ distinct} \quad \mathsf{pn}(C) \subseteq \overline{p} \\ \forall p.x \in \overline{p.x}, p \in \overline{p} \quad \langle I, K \rangle \checkmark \text{ for each } I \in \mathsf{stats}(C) \\ C \text{ contains no runtime terms} \quad \mathsf{keys}(C) \text{ distinct} \quad \forall (l, t) \in \mathsf{keys}(C), t = \mathbf{t} \end{array}}{X(\overline{p}, \overline{p.x}) = C \checkmark} \text{ C-WF-DEF} \\
\\
\frac{\begin{array}{c} \langle l, \tau : X(\overline{q}, \overline{a}), K \rangle \checkmark \quad (X(q_1, \dots, q_n, q^1.x_1, \dots, q^m.x_m) = C') \in \mathcal{C} \\ \{r_1, \dots, r_k\} \subseteq \{p_1, \dots, p_n\} \quad \forall i \leq k, j \leq n \text{ if } r_i = p_j \text{ then } \llbracket C \rrbracket_{r_i} = \llbracket C' \rrbracket_{q_j} \end{array}}{\langle l, \tau : r_1, \dots, r_k. X(p_1, \dots, p_n, a_1, \dots, a_m) \{C\}, K \rangle \checkmark} \text{ C-WF-CALLING} \\
\\
\begin{array}{ll} \mathsf{pn}(0) = \emptyset & \\ \mathsf{pn}(I; C) = \mathsf{pn}(I) \cup \mathsf{pn}(C) & \\ \mathsf{pn}(\{C\}) = \mathsf{pn}(C) & \\ \mathsf{pn}(l, t : p.e \rightarrow val q.x) = \{p, q\} & \\ \mathsf{pn}(l, t : p \rightsquigarrow q.x) = \{q\} & \\ \mathsf{pn}(l, t : p \rightarrow q[L]) = \{p, q\} & \\ \mathsf{pn}(l, t : p \rightsquigarrow q[L]) = \{q\} & \\ \mathsf{pn}(l, t : val p.x = e) = \{p\} & \\ \mathsf{pn}(l, t : if e@p then C_1 else C_2) = & \\ \quad \{p\} \cup \mathsf{pn}(C_1) \cup \mathsf{pn}(C_2) & \\ \mathsf{pn}(l, t : X(\overline{p}, \overline{a})) = \overline{p} & \\ \mathsf{pn}(l, t : \overline{q}. X(\overline{p}, \overline{a}) \{C\}) = \overline{p} & \\ \mathsf{pn}(v@p) = \{p\} & \\ \mathsf{pn}(p.x) = \{p\} & \end{array}
\end{array}$$

 **Figure 9** Well-formedness (representative rules).

To prove these properties we need an invariant that characterizes how the rules of O_3 preserve the intuition from Section 2. For example, consider the following configurations:

$$\langle l, \tau_0 : p \rightsquigarrow q.x, \Sigma, \{p \mapsto \emptyset, q \mapsto \emptyset\} \rangle \tag{1}$$

$$\langle l, \tau_0 : p.e \rightarrow val q.x, \Sigma, \{p \mapsto \emptyset, q \mapsto \{(l, \tau, v)\}\} \rangle \tag{2}$$

$$\langle \{1, \tau : p.e \rightarrow val q.x\}; \{1, \tau : p.e' \rightarrow val q.x\}, \Sigma, \{p \mapsto \emptyset, q \mapsto \emptyset\} \rangle \tag{3}$$

$$\langle 3, \tau_0 : p.X(p, q) \{1, \tau_0 : p.e \rightarrow val q.x\}, \Sigma, \{p \mapsto \emptyset, q \mapsto \emptyset\} \rangle \tag{4}$$

Configuration (1) is not reachable because $l, \tau : p \rightsquigarrow q.x$ never occurs unless q has an undelivered message from p . Dually, configuration (2) is not reachable because p has a message in its queue that, according to the choreography, has not yet been sent. Configuration (3) is unreachable because the two instructions share the same integrity key; we will show that `nextToken` ensures such configurations never arise. Likewise, `nextToken` also forbids configuration (4), since the token of the instruction $1, \tau_0 : p.e \rightarrow val q.x$ must have been derived from the integrity key of the enclosing call $3, \tau_0 : p.X(p, q) \{ \dots \}$. Specifically, $\tau_0 \neq \text{nextToken}(3, \tau_0)$. To specify this last property, recall that tokens are represented as lists of integers $l_1 :: l_2 :: \dots$. We say (l_1, t_1) is a *prefix* of (l_2, t_2) – written $(l_1, t_1) \prec (l_2, t_2)$ – if the list $l_1 :: t_1$ is a prefix of $l_2 :: t_2$ and that the keys are *disjoint* if neither is a prefix of the other.

Following convention, we formalize the properties of reachable configurations by defining which configurations and procedures are *well-formed*. Figure 9 highlights the most interesting rules that define well-formedness, where \checkmark reads “well-formed” – the rest can be found in the full version of this paper [27]. In particular, well-formedness ensures that:

1. (C-WF-SEND) $l, \tau : p \rightsquigarrow q.x$ occurs in C if and only if $(l, \tau, v) \in K(q)$ for some v .
2. (C-WF-SELECT) $l, \tau : p \rightsquigarrow q[L]$ occurs in C if and only if $(l, \tau, L) \in K(q)$.
3. (C-WF-DEF) Each I in C has a distinct integrity key l, t , where t is not a placeholder.
4. (C-WF-CALLING) If the integrity key of I is a prefix of the integrity key of I' then I is a communication-in-progress $l, t : \bar{p}. X(\bar{p}, \bar{a}) \{ C' \}$ and I' is in C' .

Well-formedness also guarantees other properties seen in other choreography models, e.g., that procedures contain no free variables and that processes waiting to enter a procedure have the same local behaviour in the original procedure body and the current choreography [26]. As in prior work [26, 13], the latter check is made by using endpoint projection ($\llbracket C \rrbracket_p$), which returns the local behaviour of a process in a choreography and is defined later in Section 4.3.

► **Theorem 1** (Preservation). *If $\langle C, \Sigma, K \rangle$ is well-formed and $\langle C, \Sigma, K \rangle \xrightarrow{P} \langle C', \Sigma', K' \rangle$, then $\langle C', \Sigma', K' \rangle$ is well-formed.*

Proof. By induction on the rules of \xrightarrow{P} . We focus on the rules for communication and procedure invocation.

C-SEND replaces $l, \tau : p.e \rightarrow \text{val } q.x$ with $l, \tau : p \rightsquigarrow q.x$ and adds a message (l, τ, v) . By the induction hypothesis, (l, τ, v) is not already in K .

C-RECV eliminates $l, \tau : p \rightsquigarrow q.x$ and removes a message (l, τ, v) . Since each instruction has a distinct integrity key by hypothesis, no other $l, \tau : p \rightsquigarrow q.x$ term occurs in C .

C-FIRST introduces new terms into the choreography by invoking the call $l_1, \tau_1 : X(\bar{p}, \bar{a})$. By the induction hypothesis, for any other instruction $l_2, t_2 : I$ in C , either (a) keys l_1, t_1 and l_2, t_2 are disjoint; or (b) $l_2, t_2 : I$ is a call-in-progress containing $l_1, \tau_1 : X(\bar{p}, \bar{a})$. In case (a), disjointness implies any instruction in the body of the procedure $C'[\bar{q}, \bar{q}.y, t \mapsto \bar{p}, \bar{p}.x, \tau']$ will also have a key that is disjoint from l_2, t_2 . In case (b), notice $\forall l, (l_2, t_2) \prec (l_1, t_1) \prec (l, \text{nextToken}(l_1, t_1))$; hence any interaction in the body has a key where (l_2, t_2) is a prefix. ◀

► **Theorem 2** (Deadlock-Freedom). *If $\langle C, \Sigma, K \rangle$ is well-formed, then either $C \equiv 0$ or $\langle C, \Sigma, K \rangle \xrightarrow{P} \langle C', \Sigma', K' \rangle$ for some p, C', Σ', K' .*

Proof. By induction on the structure of C , making use of the full definition of well-formedness [27]. In each case, we observe the first instruction I of C can always be executed. For instance, if $I \equiv l, \tau : p.e \rightarrow \text{val } q.x$ then the C-SEND rule can be applied because well-formedness implies e has no free variables. If $I \equiv l, \tau : p \rightsquigarrow q.x$, there must be a message $(l, \tau, v) \in K(q)$ because the configuration is well-formed. The other cases follow similarly. ◀

We end this section with a formalization of communication integrity. Consider the buggy execution in Figure 2: in a model without integrity keys, the execution reaches a configuration

$$\langle s \rightsquigarrow c.txt; s \rightsquigarrow c.key; \dots, \Sigma, c \mapsto v_{key}, v_{txt} \rangle,$$

where v_{key} is the value produced by $ks.getKey()$ and v_{txt} is the value produced by $cs.getText()$. A CIV occurs if the configuration can make a transition that consumes $s \rightsquigarrow c.txt$ and v_{key} together, binding $c.txt$ to v_{key} . We therefore want to ensure:

- There is only one way a communication-in-progress instruction can be consumed; and
- The instruction is consumed together with the correct message.

► **Definition 3** (Send/receive transitions). A send transition $\langle C, \Sigma, K \rangle \xrightarrow{P} \langle C', \Sigma', K' \rangle$ is a transition with a derivation that ends with an application of C-SEND. Likewise, a receive transition is a transition with a derivation that ends with C-RECV.

► **Theorem 4** (Communication Integrity). Let $e = c_0 \xrightarrow{p_1} \dots \xrightarrow{p_{k+1}} c_{k+1}$ be an execution ending with a send transition $c_k \xrightarrow{p} c_{k+1}$, which produces instruction $l, \tau : p \rightsquigarrow q.x$ and message m . Let $e' = c_0 \xrightarrow{p_1} \dots \xrightarrow{p_n} c_n$ ($n > k$) be an execution extending e , where $l, \tau : p \rightsquigarrow q.x$ has not yet been consumed. Then there is at most one receive transition $c_n \xrightarrow{q} c_{n+1}$ consuming $l, \tau : p \rightsquigarrow q.x$. Namely, it is the transition that consumes $l, \tau : p \rightsquigarrow q.x$ and m together.

Proof. By definition of C-SEND, m has the form (l, τ, v) . By definition of C-RECV, if there exists a transition $c_n \rightarrow c_{n+1}$ that consumes $l, \tau : p \rightsquigarrow q.x$, then the transition also consumes a message (l, τ, v') , for some v' . It therefore suffices to show the message (l, τ, v') is unique and that $v' = v$. This follows by induction on the length m of the extension:

- *Base case:* Well-formedness implies there is no message (l, τ, v') in c_k . Hence the message (l, τ, v) in c_{k+1} is unique.
- *Induction step:* Observe that the transition $c_m \rightarrow c_{m+1}$ cannot remove (l, τ, v) ; this would require consuming $l, \tau : p \rightsquigarrow q.x$, which cannot happen in e' by hypothesis. Also observe that the transition cannot add a new message with integrity key (l, τ) ; this would require consuming an instruction $l, \tau : p'.e \rightarrow \text{val } q.x'$, which cannot exist in c_m by well-formedness. Hence (l, τ, v) is unique in c_{m+1} . ◀

4 Process Model and Endpoint Projection

4.1 Syntax

Figure 10 presents the syntax for out-of-order processes. A term $p[P]$ is a process named p with behavior P . Networks, ranged over by N, M , are parallel compositions of processes. Compared to prior work, certain process instructions need to be annotated with integrity keys (for instance, message send $p!_{l,t} e$ and procedure call $l, t : X(\bar{p}, \bar{a})$). In addition, when receiving a message it is no longer necessary to specify a sender – it suffices to write $?_{l,t} x; P$ instead of the more traditional $p?_{l,t} x; P$ – because integrity keys functionally determine the variable to which the message payload should be bound.

$\mathcal{P} ::= \{X_i(\bar{p}_i, \bar{x}_i) = C_i\}_{i \in \mathcal{I}}$	(decls)	
$P, Q ::= I; P$	(seq)	$\{P\}$ (block)
0	(end)	
$I ::= p!_{l,t} e$	(send)	$?_{l,t} x$ (receive)
$\text{val } x = e$	(expr)	$p \oplus_{l,t} L$ (choice)
$\&\{(l, \tau, L_i) \Rightarrow P_i\}_{i \in \mathcal{I}}$	(branch)	if e then P else Q (cond)
$l, t : X(\bar{p}, \bar{a})$	(call)	
$e ::= f(\bar{e})$	(app)	a (atom)
$a ::= x$	(var)	v (val)
$N, M ::= p[P]$	(proc)	$(N \mid M)$ (par)

Figure 10 Syntax for out-of-order processes.

4.2 Semantics

The semantics for out-of-order processes appears in Figure 11. It is a labelled transition system on *process configurations* $\langle N, \Sigma, K \rangle$, where N is a network and Σ, K have the same meaning as in Section 3.2. We also assume an implicit set of procedure declarations \mathcal{P} .

The transition rules of Figure 11 are similar to prior work. P-SEND adds a message (l, τ, v) to the undelivered messages of q , whereas P-RECV removes the message and substitutes it into the body of the process. Similarly, P-SELECT adds (l, τ, L) to the message set and P-ONSELECT selects a branch from the set of options $\&\{(l_j, \tau_j, L_j) \Rightarrow P_j\}_{j \in \mathcal{J}}$. P-CALL invokes a procedure, locally computing the next token and substituting the body of the procedure into the process. Rules P-COMPUTE, P-IF, and P-PAR are standard.

The key novelty of out-of-order processes is the P-DELAY rule, which allows a process to perform instructions in *any* order, up to data- and control-dependencies. The latter implies processes cannot evaluate instructions nested within an if or $\&$ -expression.

4.3 Endpoint Projection

Figure 13 defines the *endpoint projection* (EPP) $\llbracket C \rrbracket$ of a choreography C , translating it into a network. The rules follow from simple modifications to the textbook definition of EPP [26]. Projecting a conditional on a process that does not evaluate the guard uses the auxiliary partial operator \sqcup , which produces a term that can react to the different branches by receiving different selections (this is standard).

Figure 12 shows networks projected from the choreographies of Figure 7. Notice the choreographic procedures *BuyItem* and *StreamIt* are each split into two process procedures – one for each role. Communications in the choreography are, as usual, projected into send and receive instructions. Conditionals in the choreography are projected into an if-instruction at one process and a branch-instruction at the other processes awaiting its decision.

Below we formulate the hallmark *EPP Theorem*, which states that a choreography C and its projection $\llbracket C \rrbracket$ evolve in lock-step, up to the usual (\sqsupseteq) relation from Montesi [26] (given in Figure 14). Importantly, we update the theorem to restrict our attention to *well-formed* networks and choreographies. We say that a network N is well-formed if the keys in each process are distinct, i.e., $\text{keys}(P)$ is distinct for each $p[P]$ in N ($\text{keys}(P)$ is given in Figure 14). The restriction allows us to ignore processes such as $p[\&\{(1, \tau, L) \Rightarrow P_1\}; \&\{(1, \tau, L) \Rightarrow P_2\}]$, which could only be projected from a choreography where two distinct instructions have the same integrity key $(1, \tau)$. This leads to the following lemma:

► **Lemma 5.** *Let C, Q be well-formed. If $Q \sqsupseteq \llbracket C \rrbracket_q$ then $\text{keys}(Q) \supseteq \text{keys}_q(C)$, where*

$$\text{keys}_q(C) = [(l, t) \mid (l, t : p \rightsquigarrow q.x) \in \text{stats}(C)], [(l, t) \mid (l, t : p.e \rightarrow \text{val } q.x) \in \text{stats}(C)].$$

The key difficulty of proving the EPP Theorem was finding the right definition of well-formedness (Theorems 1 and 4). With the definition established, the entire proof follows directly from textbook induction principles (*c.f.* [26]). We sketch the proof in the full version of this paper [27].

► **Theorem 6 (EPP Theorem).** *Let $\langle C, \Sigma, K \rangle$ be a well-formed configuration.*

1. (Completeness) *If $\langle C, \Sigma, K \rangle \xrightarrow{p} \langle C', \Sigma', K' \rangle$ then $\langle \llbracket C \rrbracket, \Sigma, K \rangle \xrightarrow{p} \langle N', \Sigma', K' \rangle$ for some well-formed N' where $N' \sqsupseteq \llbracket C' \rrbracket$.*
2. (Soundness) *If $\langle N, \Sigma, K \rangle \xrightarrow{r} \langle N', \Sigma', K' \rangle$ for some well-formed N where $N \sqsupseteq \llbracket C \rrbracket$, then $\langle C, \Sigma, K \rangle \xrightarrow{p} \langle C', \Sigma', K' \rangle$ for some C' where $N' \sqsupseteq \llbracket C' \rrbracket$.*

$$\begin{array}{c}
 \frac{\Sigma(\mathbf{p}) \vdash e \Downarrow (v, \sigma) \quad M = K(\mathbf{q}) \uplus \{(l, \tau, v)\}}{\langle \mathbf{p}[\mathbf{q} !_{l, \tau} e; P], \Sigma, K \rangle \xrightarrow{\mathbf{p}} \langle \mathbf{p}[P], \Sigma[\mathbf{p} \mapsto \sigma], K[\mathbf{q} \mapsto M] \rangle} \text{P-SEND} \\
 \\
 \frac{(l, \tau, v) \in K(\mathbf{q}) \quad M = K(\mathbf{q}) \setminus \{(l, \tau, v)\}}{\langle \mathbf{q}[?_{l, \tau} x; Q], \Sigma, K \rangle \xrightarrow{\mathbf{q}} \langle \mathbf{q}[Q[x \mapsto v]], \Sigma, K[\mathbf{q} \mapsto M] \rangle} \text{P-RECV} \\
 \\
 \frac{M = K(\mathbf{q}) \cup \{(l, \tau, L)\}}{\langle \mathbf{p}[\mathbf{q} \oplus_{l, \tau} L; P], \Sigma, K \rangle \xrightarrow{\mathbf{p}} \langle \mathbf{p}[P], \Sigma, K[\mathbf{q} \mapsto M] \rangle} \text{P-SELECT} \\
 \\
 \frac{K(\mathbf{q}) = \{(l_i, \tau, L_i)\} \cup M \quad i \in \mathcal{I}}{\langle \mathbf{q}[\&\{(l_j, \tau, L_j) \Rightarrow Q_j\}_{j \in \mathcal{I}}; Q], \Sigma, K \rangle \xrightarrow{\mathbf{q}} \langle \mathbf{q}[\{Q_i\}; Q], \Sigma, K[\mathbf{q} \mapsto M] \rangle} \text{P-ONSELECT} \\
 \\
 \frac{\Sigma(\mathbf{p}) \vdash e \Downarrow (v, \sigma)}{\langle \mathbf{p}[\mathbf{val} x = e; P], \Sigma, K \rangle \xrightarrow{\mathbf{p}} \langle \mathbf{p}[P[x \mapsto v]], \Sigma[\mathbf{p} \mapsto \sigma], K \rangle} \text{P-COMPUTE} \\
 \\
 \frac{\Sigma(\mathbf{p}) \vdash e \Downarrow v \quad \text{if } v = \mathbf{true} \text{ then } i = 1 \text{ else } i = 2}{\langle \mathbf{p}[\mathbf{if } e \mathbf{then } P_1 \mathbf{else } P_2; P], \Sigma, K \rangle \xrightarrow{\mathbf{p}} \langle \mathbf{p}[\{P_i\}; P], \Sigma, K \rangle} \text{P-IF} \\
 \\
 \frac{\langle \mathbf{p}[P_1], \Sigma, K \rangle \xrightarrow{\mathbf{p}} \langle \mathbf{p}[P'_1], \Sigma', K' \rangle}{\langle \mathbf{p}[\{P_1\}; P_2], \Sigma, K \rangle \xrightarrow{\mathbf{p}} \langle \mathbf{p}[\{P'_1\}; P_2], \Sigma', K' \rangle} \text{P-BLOCK} \\
 \\
 \frac{\langle \mathbf{p}[P], \Sigma, K \rangle \xrightarrow{\mathbf{p}} \langle \mathbf{p}[P'], \Sigma', K' \rangle}{\langle \mathbf{p}[I; P], \Sigma, K \rangle \xrightarrow{\mathbf{p}} \langle \mathbf{p}[I; P'], \Sigma', K' \rangle} \text{P-DELAY} \\
 \\
 \frac{(X(\bar{q}, \bar{y}) = Q) \in \mathcal{P} \quad \text{nextToken}(l, \tau) = \tau'}{\langle \mathbf{p}[l, \tau : X(\bar{p}, \bar{a}); P], \Sigma, K \rangle \xrightarrow{\mathbf{p}} \langle \mathbf{p}[\{Q[\bar{q}, \bar{y}, \mathbf{t} \mapsto \bar{p}, \bar{a}, \tau']\}; P], \Sigma, K \rangle} \text{P-CALL} \\
 \\
 \frac{\langle N, \Sigma, K \rangle \xrightarrow{\mathbf{p}} \langle N', \Sigma', K' \rangle}{\langle N \mid M, \Sigma, K \rangle \xrightarrow{\mathbf{p}} \langle N' \mid M, \Sigma', K' \rangle} \text{P-PAR}
 \end{array}$$

 **Figure 11** Semantics for out-of-order processes.

```

BuyItem1(b) =
?1,t itemID;
val item? = sell(itemID);
b!3,t item?

BuyItem2(s, itemID) =
s!1,t itemID;
?3,t item?
seller[4, τ0 : BuyItem1(buyer1);
      5, τ0 : BuyItem1(buyer2)] |
buyer1[4, τ0 : BuyItem2(seller, 123)] |
buyer2[5, τ0 : BuyItem2(seller, 543)]

```

(a)


```

StreamIt1(c) =
c!1,t produce();
if (itemsLeft() > 0) then
  c ⊕4,t MORE; 5, t : StreamIt1(c)
else c ⊕6,t DONE

StreamIt2(p) =
?1,t x; val z = consume(x);
& {(4, t, MORE) ⇒ 5, t : StreamIt2(p),
      (6, t, DONE) ⇒ 0}
p1[7, τ0 : StreamIt1(c)] |
p2[8, τ0 : StreamIt1(c)] |
c[7, τ0 : StreamIt2(p); 8, τ0 : StreamIt2(p)]

```

(b)

Figure 12 Processes projected from Figure 7.

5 A Non-Blocking Communication API for Choral

In this section, we show how the ideas in O_3 can be applied in practice. To this end, we consider Choral [16]: a state-of-the-art choreographic programming language based on Java. Choral is designed to support real-world programming and interoperate with Java, so it is much more sophisticated than our minimalistic theory. Data locations in Choral are lifted to the type level and communication is expressed by invoking methods of *channel* objects.

Choral’s intended programming model consists of sequential processes that block to receive messages. However, to improve performance programmers can use Java’s `CompletableFuture` API, thereby introducing intraprocess concurrency and out-of-order execution. This breaks the programming model and introduces CIVs (cf. Section 2) that could cause crashes or silent memory corruption. Motivated by our formal model, we developed *Ozone*: an API for Choral programmers to safely mix choreographies with futures. In the remainder of this section, we introduce Choral and Ozone and we show how programmers can mix choreographies with futures achieve significant speedups in practical applications.

5.1 Concurrent Messages

We introduce the Ozone API with an implementation of the choreographic procedure from Figure 2. The implementation is shown in Figure 15, which defines a class called `ConcurrentSend` parameterized by four roles (i.e. process parameters): `KS`, `CS`, `S`, and `C`. In this class, the `start` method implements the procedure itself. As in our formal model, the procedure is parameterized by distributed data: On line 3, parameter `key` is a `String` located at `KS`; `txt` is a `String` located at `CS`; and `client` is a `Client` object at `C`, representing the client’s user interface. The `start` procedure is also parameterized by session tokens, which we introduced in Figure 15, on line 4. The parameter `Token@(KS, CS, S, C) tok` is

$$\begin{aligned}
 \llbracket \mathcal{C} \rrbracket &= \bigcup_{i \in \mathcal{I}} \llbracket X_i(\bar{\mathbf{p}}, \bar{\mathbf{p}}.\bar{x}) = C_i \rrbracket \\
 \llbracket X_i(\bar{\mathbf{p}}, \bar{\mathbf{p}}.\bar{x}) = C_i \rrbracket &= \{X_{i,j}(\bar{\mathbf{p}} \setminus \mathbf{p}_j, \llbracket \bar{\mathbf{p}}.\bar{x} \rrbracket_{\mathbf{p}_j}) = \llbracket C_i \rrbracket_{\mathbf{p}_j} \mid \bar{\mathbf{p}} = \mathbf{p}_1, \dots, \mathbf{p}_n, j \leq n\} \\
 \llbracket l, t : \mathbf{p}.e \rightarrow \mathbf{val} \quad \mathbf{q}.x; \quad C \rrbracket_r &= \begin{cases} \mathbf{q} !_{l,t} e; \llbracket C \rrbracket_r & \text{if } r = \mathbf{p} \\ ?_{l,t} x; \llbracket C \rrbracket_r & \text{if } r = \mathbf{q} \\ \llbracket C \rrbracket_r & \text{otherwise} \end{cases} \\
 \llbracket l, t : \mathbf{p} \rightsquigarrow \mathbf{q}.x; \quad C \rrbracket_r &= \begin{cases} ?_{l,t} x; \llbracket C \rrbracket_r & \text{if } r = \mathbf{q} \\ \llbracket C \rrbracket_r & \text{otherwise} \end{cases} \\
 \llbracket l, t : \mathbf{val} \quad \mathbf{p}.x = e; \quad C \rrbracket_r &= \begin{cases} \mathbf{val} x = \llbracket e \rrbracket_r, \llbracket C \rrbracket_r & \text{if } r = \mathbf{p} \\ \llbracket C \rrbracket_r & \text{otherwise} \end{cases} \\
 \llbracket l, t : \mathbf{p} \rightarrow \mathbf{q}[L]; \quad C \rrbracket_r &= \begin{cases} \mathbf{q} \oplus_{l,t} \llbracket e \rrbracket_r; \llbracket C \rrbracket_r & \text{if } r = \mathbf{p} \\ \&\{(l, t, L) \Rightarrow \llbracket C \rrbracket_r\} & \text{if } r = \mathbf{q} \\ \llbracket C \rrbracket_r & \text{otherwise} \end{cases} \\
 \llbracket l, t : \mathbf{p} \rightsquigarrow \mathbf{q}[L]; \quad C \rrbracket_r &= \begin{cases} \&\{(l, t, L) \Rightarrow \llbracket C \rrbracket_r\} & \text{if } r = \mathbf{q} \\ \llbracket C \rrbracket_r & \text{otherwise} \end{cases} \\
 \llbracket l, t : \mathbf{if} \; e @ \mathbf{p} \; \mathbf{then} \; C_1 \; \mathbf{else} \; C_2; \quad C \rrbracket_r &= \begin{cases} \mathbf{if} \; \llbracket e \rrbracket_r \; \mathbf{then} \; \llbracket C_1 \rrbracket_r \; \mathbf{else} \; \llbracket C_2 \rrbracket_r; \llbracket C \rrbracket_r & \text{if } r = \mathbf{p} \\ \llbracket C_1 \rrbracket_r \sqcup \llbracket C_2 \rrbracket_r; \llbracket C \rrbracket_r & \text{if } r \in \mathbf{pn}(C_1, C_2) \setminus \mathbf{p} \\ \llbracket C \rrbracket_r & \text{otherwise} \end{cases} \\
 \llbracket l, t : X_i(\bar{\mathbf{p}}, \bar{a}); \quad C \rrbracket_r &= \begin{cases} l, t : X_{i,j}(\bar{\mathbf{p}} \setminus \mathbf{p}_j, \llbracket \bar{a} \rrbracket_{\mathbf{p}_j}); \llbracket C \rrbracket_{\mathbf{p}_j} & \text{if } r = \mathbf{p}_j \text{ where } \bar{\mathbf{p}} = \mathbf{p}_1, \dots, \mathbf{p}_n \\ \llbracket C \rrbracket_r & \text{otherwise} \end{cases} \\
 \llbracket l, t : \bar{\mathbf{q}}. \quad X_i(\bar{\mathbf{p}}, \bar{a}) \{ C_1 \}; \quad C_2 \rrbracket_r &= \begin{cases} l, t : X_{i,j}(\bar{\mathbf{p}} \setminus \mathbf{p}_j, \llbracket \bar{a} \rrbracket_{\mathbf{p}_j}); \llbracket C_2 \rrbracket_{\mathbf{p}_j} & \text{if } r \in \bar{\mathbf{q}} \text{ and } r = \mathbf{p}_j \\ \llbracket C_1; C_2 \rrbracket_r & \text{if } r \in \bar{\mathbf{p}} \setminus \bar{\mathbf{q}} \\ \llbracket C_2 \rrbracket_r & \text{otherwise} \end{cases} \\
 \llbracket \{ C_1 \}; \quad C_2 \rrbracket_r &= \{ \llbracket C_1 \rrbracket_r \}; \llbracket C_2 \rrbracket_r & \llbracket v @ \mathbf{p} \rrbracket_r = \begin{cases} v & \text{if } r = \mathbf{p} \\ \perp & \text{otherwise} \end{cases} \\
 \llbracket a_1, \dots, a_n \rrbracket_r &= \llbracket a_1 \rrbracket_r, \dots, \llbracket a_n \rrbracket_r \\
 \llbracket f(e_1, \dots, e_n) \rrbracket_r &= f(\llbracket e_1 \rrbracket_r, \dots, \llbracket e_n \rrbracket_r) & \llbracket \mathbf{p}.x \rrbracket_r = \begin{cases} x & \text{if } r = \mathbf{p} \\ \perp & \text{otherwise} \end{cases} \\
 (\&\{(l_i, \tau_i, L_i) \Rightarrow P_i\}_{i \in \mathcal{I}}) \sqcup (\&\{(l_j, \tau_j, L_j) \Rightarrow P_j\}_{j \in \mathcal{J}}) &= \&\{(l_k, \tau_k, L_k) \Rightarrow P_k\}_{k \in \mathcal{I} \cup \mathcal{J}} \\
 &\text{if } \{L_i : i \in \mathcal{I}\} \# \{L_j : j \in \mathcal{J}\}
 \end{aligned}$$

 **Figure 13** Endpoint projection.

$0 \sqsupseteq 0$	
$(P_1; P_2) \sqsupseteq (Q_1; Q_2)$	if $P_i \sqsupseteq Q_i$ for $i = 1, 2$
$(\text{if } e \text{ then } P_1 \text{ else } P_2) \sqsupseteq (\text{if } e \text{ then } Q_1 \text{ else } Q_2)$	if $P_i \sqsupseteq Q_i$ for $i = 1, 2$
$I_1 \sqsupseteq I_2$	if $I_1 = I_2$ or $I_1 = I_1 \sqcup I_2$
$\text{keys}(0) = \epsilon$	
$\text{keys}(I; P) = \text{keys}(I), \text{keys}(P)$	$\text{keys}(\&\{(l_i, \tau_i, L_i) \Rightarrow P_i\}_{i \in \mathcal{I}}) =$ $[(l_i, \tau_i) \mid i \in \mathcal{I}], [\text{keys}(P_i) \mid i \in \mathcal{I}]$
$\text{keys}(\mathbf{p} !_{l,t} e) = (l, t)$	$\text{keys}(\text{if } e \text{ then } P_1 \text{ else } P_2) = \text{keys}(P_1), \text{keys}(P_2)$
$\text{keys}(\mathbf{?}_{l,t} x) = (l, t)$	$\text{keys}(l, t : X(\bar{p}, \bar{a})) = (l, t)$
$\text{keys}(\mathbf{val} x = e) = \epsilon$	$\text{keys}(\{P_1\}; P_2) = \text{keys}(P_1), \text{keys}(P_2)$
$\text{keys}(\mathbf{p} \oplus_{l,t} L) = (l, t)$	

Figure 14 Auxiliary definitions for the EPP Theorem (\sqsupseteq and keys).

```

1  public class ConcurrentSend@(KS, CS, S, C) {
2      public void start(
3          String@KS key, String@CS txt, Client@C client,
4          Token@(KS, CS, S, C) tok,
5          AsyncChannel@(KS, S) ch1, AsyncChannel@(CS, S) ch2, AsyncChannel@(S, C) ch3
6      ) {
7          // Services send data to the server.
8          CompletableFuture@S keyS = ch1.fcom(key, 1@(KS,S), tok);
9          CompletableFuture@S txtS = ch2.fcom(txt, 2@(CS,S), tok);
10
11         // Server forwards data to the client.
12         ch3.fcom(keyS, 3@(S,C), tok)
13             .thenAccept(client::decrypt);
14         ch3.fcom(txtS, 4@(S,C), tok)
15             .thenAccept(client::display);
16     }
17 }
```

Figure 15 An implementation of the choreography in Figure 2 using Choral and the Ozone API.

syntactic sugar for the parameter *list* `Token@KS tok_KS, ..., Token@C tok_C`.³ The last three parameters on line 5 are *channels*. In Choral, channels are used to communicate data from one role to another. If `ch` is a channel of type `Channel@(A,B)<T>` and `e` is an expression of type `T@A`, then the expression `ch.com(e)` is a communication that produces a value of type `T@B`.

Our main contribution in the Ozone API is a custom channel `AsyncChannel@(A,B)<T>` with a method `fcom` for safely communicating data with non-blocking semantics. The `fcom` method is similar to `com`, but with the following differences:

- Whereas `com` takes one argument, `fcom` takes three: a payload, a line number, and a session token. The latter two arguments form an integrity key, of which both the sender and receiver have a copy.

³ This syntactic sugar is provided for readability and is not currently supported by the Choral compiler. We will also use syntactic sugar for lambda expressions and omit obvious type annotations later in this section. Our actual implementation uses desugared versions of the syntax.

31:20 Ozone: Fully Out-of-Order Choreographies

```

1  public class ConcurrentSend_KS {
2      public void start(
3          String key, Token tok_KS,
4          AsyncChannel ch1
5      ) {
6          ch1.fcom(key, 1, tok_KS);
7      }
8  }
9  public class ConcurrentSend_S {
10     public void start(
11         Token tok_S, AsyncChannel ch1,
12         AsyncChannel ch2, AsyncChannel ch3
13     ) {
14         CompletableFuture keyS =
15             ch1.fcom(1, tok_S);
16         CompletableFuture txtS =
17             ch2.fcom(2, tok_S);
18
19         ch3.com(keyS, 3, tok_S);
20         ch3.com(txtS, 4, tok_S);
21     }
22 }
```

```

23  public class ConcurrentSend_CS {
24      public void start(
25          String txt, Token tok_CS,
26          AsyncChannel ch2
27      ) {
28          ch2.fcom(txt, 2, tok_CS);
29      }
30  }
31
32  public class ConcurrentSend_C {
33      public void start(
34          Client client, Token tok_C,
35          AsyncChannel ch3
36      ) {
37          ch3.fcom(3, tok_C)
38              .thenAccept(client::decrypt);
39          ch3.fcom(4, tok_C)
40              .thenAccept(client::display);
41      }
42 }
```

 **Figure 16** Endpoint projection of Figure 15.

- When the receiver B executes a `com` instruction, its thread becomes blocked until the value (of type `T@B`) has been delivered. In contrast, `fcom` creates a Java *future* (of type `CompletableFuture@B<T>`) which is a placeholder at B that will hold a value of type T once the message is delivered. Instead of blocking, `fcom` immediately returns that future to the calling thread. The thread can then assign a callback to handle the message and proceed with other useful work.

Lines 8 and 9 of Figure 15 show `fcom` being used to transport `key` and `txt` to the server S. The expression `1@(KS, S)` is sugar for the list `1@KS, 1@S` and we assume the replicated value `tok` is expanded into the list `tok_KS, tok_S`. Thus both sender and receiver pass integrity keys as arguments to `fcom`.

Lines 12-15 of Figure 15 show how the server S and client C use the future values. On line 12, the server uses an overloaded version of `fcom` that takes `CompletableFuture@S` instead of `T@S`. The method assigns to the future a callback, which forwards the result to the client once the future has been completed. The result of `fcom` on line 12 is a `CompletableFuture@C`, to which the client binds a callback on line 13: when the key from S finally arrives at C, the client will proceed to invoke the method `client.decrypt` with the key as an argument. Lines 14 and 15 do the same, but with the value of `txt`. As we will see below, the values of `key` and `txt` can arrive at the client in any order, so the callbacks on lines 13 and 15 can execute in any order – even in parallel.

5.1.1 Endpoint projection

By running the Choral compiler, `ConcurrentSend@(KS,CS,S,C)` is *projected* to generate four Java classes, shown in Figure 16. Each class implements the behavior of its corresponding role. For example, `ConcurrentSend_KS` implements the behavior of KS. Its `start` method is parameterized by: `key`, which corresponds to the `key` in Figure 15; `tok_KS`, the copy of the token `tok` belonging to KS; and `ch1`, a channel endpoint that connects KS to S. Following the reasoning in Figure 13, these behaviors will not exhibit deadlocks or communication integrity errors when composed (assuming the implementations of Choral and Ozone are correct).

```

1  public class ConcurrentClients@{KS, CS, S, C1, C2} {
2      public void start(
3          AsyncChannel@{KS, S} ch1, AsyncChannel@{CS, S} ch2,
4          AsyncChannel@{S, C1} ch3, AsyncChannel@{S, C2} ch4,
5          KeyService@{KS} keyService, ContentService@{CS} contentService,
6          Client@{C1} client1, String@{KS, CS} clientID1,
7          Client@{C2} client2, String@{KS, CS} clientID2,
8          Token@{KS, CS, S, C1, C2} tok
9      ) {
10         (new ConcurrentSend2()).start(ch1, ch2, ch3,
11             keyService.getKey(clientID1), contentService.getContent(clientID1),
12             client1, tok.nextToken(0@{KS,CS,S,C1}) );
13
14         (new ConcurrentSend2()).start(ch1, ch2, ch4,
15             keyService.getKey(clientID2), contentService.getContent(clientID2),
16             client2, tok.nextToken(1@{KS,CS,S,C2}) );
17     }
18 }
```

Figure 17 A Choral choreography invoking ConcurrentSend2.

```

1  public class ConcurrentClients_KS {
2      public void start( ... ) {
3          (new ConcurrentSend2_KS()).start(ch1,
4              keyService.getKey(clientID1),
5              tok.next(0));
6
7          (new ConcurrentSend2_KS()).start(ch1,
8              keyService.getKey(clientID2),
9              tok.next(1));
10     }
11 }
```

```

12  public class ConcurrentClients_S {
13      public void start( ... ) {
14          (new ConcurrentSend2_S()).start(
15              ch1, ch2, ch3, tok.next(0));
16
17          (new ConcurrentSend2_S()).start(
18              ch1, ch2, ch3, tok.next(1));
19      }
20 }
```

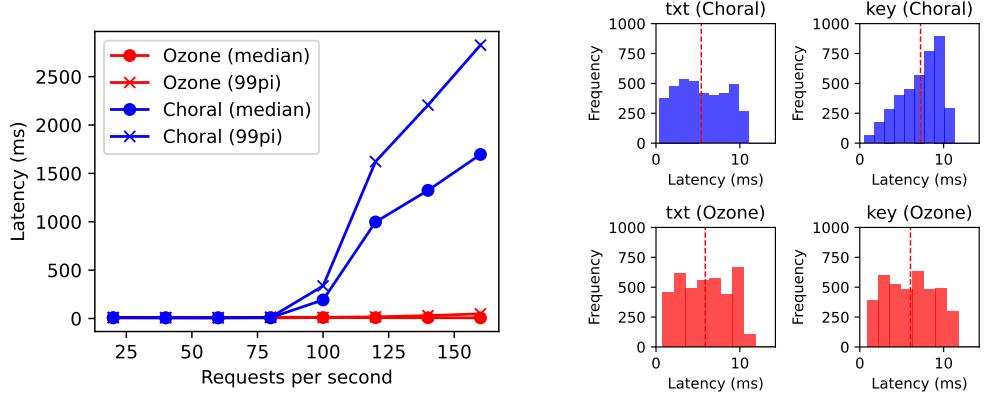
Figure 18 Endpoint projection of Figure 17 (representative examples).

Let us see how integrity keys prevent CIVs in Figure 16. Notice that the Choral instruction `CompletableFuture@S keyS = ch1.fcom(key, 1@{KS,S}, tok);` on line 8 of Figure 15 is projected into two instructions:

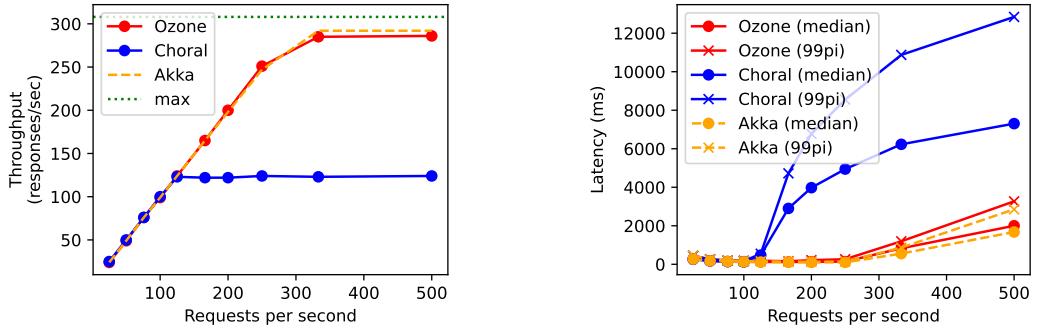
1. `ch1.fcom(key, 1, tok_KS)` at the sender KS; and
2. `CompletableFuture keyS = ch1.fcom(1, tok_S)` at the receiver S.

The former instruction is parameterized by a payload and an integrity key and produces nothing. The latter instruction is parameterized only by an integrity key (with no payload) and produces a future. When KS sends `key` to S, it combines the payload with integrity key `(1, tok_KS)`. Dually, S creates a future that will only be completed when a message with the integrity key `(1, tok_S)` is received. Since `tok_KS` and `tok_S` have the same value, the send- and receive-operations are guaranteed to match.

On lines 14-17 of Figure 16, the server S sets listeners for `key` and `txt`. On lines 19-20, S schedules the values to be forwarded to C; notice that even with FIFO channels, `key` and `txt` may arrive in any order. Consequently, S may forward their values to C in any order. On lines 37-40 of Figure 16, the client creates futures to hold the values of `key` and `txt` and sets callbacks to be invoked when the values arrive. Here we see the importance of integrity keys: the client uses `(3, tok_C)` and `(4, tok_C)` to disambiguate the `key` message from the `txt` message. Without integrity keys, mixing Choral choreographies with Java Futures would be unsafe. As shown in Section 4.3, our solution is correct even when the underlying transport protocol can deliver messages out of order.



(a) Concurrent producers latency (lower is better). (b) Concurrent senders latency (further left is better).

Figure 19 Microbenchmark.

(a) Throughput (higher is better).

(b) Latency (lower is better).

Figure 20 Model serving.

5.2 Procedure calls

Section 5.1 showed how the line numbers in an integrity key could prevent CIVs. We now briefly show how the *tokens* in an integrity key prevent *interprocedural* CIVs. Figure 17 depicts a choreography that invokes two instances of `ConcurrentSend2`: the first instance with client `C1`, and the second instance with client `C2`. On lines 12 and 16, the roles all compute fresh tokens for each procedure they're involved in, like in our formal model; the syntax `tok.nextToken(0@KS,CS,S,C1)` is sugar for `tok_KS.nextToken(0@KS), ..., tok_C1.nextToken(0@C1)`, and the method `t.nextToken(l)` implements the function `nextToken(l,t)`. These fresh tokens ensure that, even if messages from `KS` to `S` are delivered out of order there is no chance that messages from the first procedure invocation will be confused for messages from the second invocation.

5.3 Evaluation

We evaluated Ozone with microbenchmarks based on Figures 1 and 2 and with a model serving benchmark from Wang et al [34]. The experiments were carried out on a six-node Linux cluster, with two Intel Xeon Gold 6130 CPUs and 384 GB of memory per node and an average bandwidth of 15 Gbps.

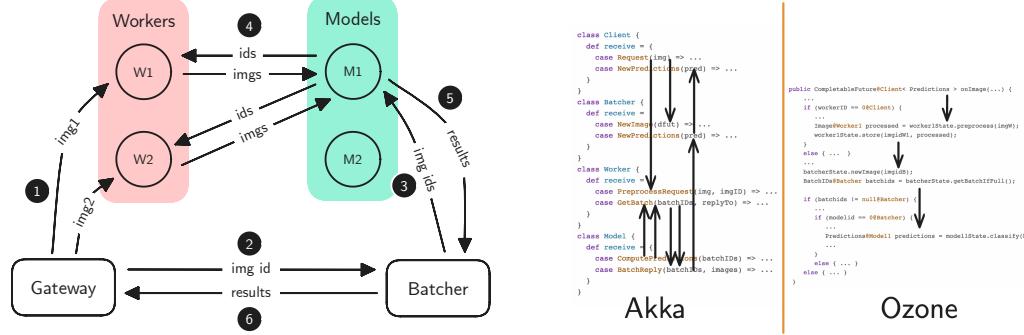


Figure 21 Architecture for the image classification pipeline.

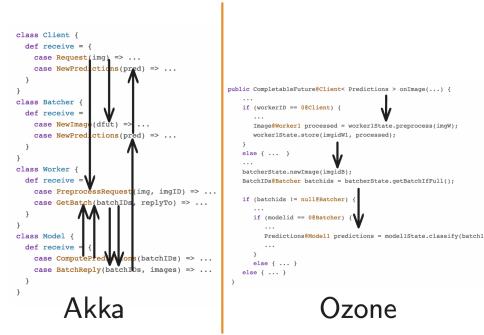
Figure 22 Control flow comparison between hand-written Akka processes and Ozone.

The first microbenchmark is a version of Figure 1 from the introduction. Each producer iteratively invokes the choreography at a fixed rate and, in response to each request, the server simulates computation by sleeping for 0–5 milliseconds. Figure 19a shows the median and 99th percentile latency for server responses to worker requests. In the Choral implementation, the server quickly becomes a bottleneck because of its fixed processing order: a request from p_1 must be handled before a request from p_2 , and both requests must be handled for the i -th iteration before they can be handled for the $(i+1)$ -th iteration. In the Ozone implementation, requests from different producers can be handled out of order and producers can start a new instance of the choreography without waiting for the second one to complete. Consequently the server spends less time waiting for requests, so it can handle much higher request rates.

The second microbenchmark is a version of Figure 2 from Section 2, in which the server sends messages to the microservices ks and cs and forwards their responses to the client. Each microservice takes 0–5 milliseconds to compute its response. The latency histogram for the Choral implementation (top) shows how the time for the client to receive txt depends on the time to compute txt , but the time to receive key depends on *both* the time to compute txt and the time to compute key . In contrast, the Ozone implementation (bottom) allows the server to forward key to the client without waiting for txt – thereby reducing the average latency for key by more than 30%.

To measure the impact of Ozone on a realistic application, we ported the image classification pipeline of Wang et al [34] to Choral (Figure 21). In this pipeline, images are received by a Gateway that performs load balancing and forwards the images to a pair of Worker services for preprocessing. A Batcher service collects requests and sends them as a batch to a Model service, which fetches the processed images and performs image classification. This architecture allows applications to harness *intra*-GPU parallelism by increasing the batch size (at the cost of latency) and *inter*-GPU parallelism by increasing the number of Model services. Figure 20 shows the performance for Choral and Ozone implementations of the pipeline, using sleeps to simulate computation. The plots also show the performance of an implementation in the Akka actor framework [3] and the theoretical maximum throughput of the Model services.

The Choral implementation has a bottleneck: after the Batcher sends work to a Model, it waits for the Model’s response and becomes blocked. In the Ozone implementation, the Batcher binds the Model service’s response to a `CompletableFuture` and continues receiving requests from the Gateway. Consequently, the throughput and latency for the Ozone implementation can scale with the number of requests until both Models become saturated with work. Figure 20 shows our library scales similarly to hand-written reactive processes in Akka, though the latter perform slightly better under high load because the



Akka framework is heavily optimized to handle network congestion; these same optimizations can be applied to Choral, but they are orthogonal to our present work. We conclude that our methodology can achieve good performance while providing the benefits of choreographic programming: (i) absence of bugs like deadlocks and mismatched communications (e.g., sending a message with the wrong type or at the wrong time) [16, 26, 23]; (ii) and improved readability, since control flow is easier to follow in choreographies than in processes (see Figure 22), as discussed in [32, 21, 23, 16].

6 Related Work

In early choreographic languages, the sequencing operator $I; C$ had strict sequential semantics; concurrency could only be introduced via an explicit parallel operator $C \parallel C'$ [30, 22, 7]. Explicit parallelism was later replaced by a relaxed sequencing operator $I; C$ that would allow instructions in C to be evaluated before I under certain conditions [8]. This presents the benefits of offering a simple syntax for choreographies and, at the same time, automatically inferring what can be safely executed concurrently. For these reasons, relaxed sequencing has been adopted in most recent works on choreographic programming (e.g., [18, 16, 17, 13, 20, 21]) and its textbook presentation [26]. Our present work makes the sequencing operator even more relaxed, allowing all instructions to be executed out of order, up to data- and control-dependency. To the best of our knowledge, our model is the first to support non-FIFO communication in the setting of choreographic programming.

Our work is closely related to choreographic *multicomms*: sets of communications that can be executed out of order, up to data dependency [12]. However, multicomms do not allow *computation* to be performed out of order, as in Figure 1c. Multicomms therefore do not need to address the communication integrity problem, which we focus on in this work. Relatedly, previous work investigated modeling asynchronous communication by making send actions non-blocking [8, 19, 10, 28, 18, 26], but none of considered non-blocking receive. Thus, they are not expressive enough to capture the behaviors that we are interested in here.

In terms of expressivity, there is some overlap between our model and *nondeterministic choreographies* [26], which use an explicit *choreographic choice* operator $C +_p C'$. Nondeterministic choreographies can implement the execution in Figure 1c with:

$$\left(\begin{array}{l} \text{buyer}_1.id \rightarrow \text{val seller}.id_1; \\ \text{buyer}_2.id \rightarrow \text{val seller}.id_2; \\ \dots \end{array} \right) +_{\text{seller}} \left(\begin{array}{l} \text{buyer}_2.id \rightarrow \text{val seller}.id_2; \\ \text{buyer}_1.id \rightarrow \text{val seller}.id_1; \\ \dots \end{array} \right)$$

Figure 2 can also be expressed with nondeterministic choreographies:

$$\left(\begin{array}{l} 1 : \text{cs.getText()} \rightarrow \text{val s.txt}; \\ 2 : \text{s} \rightarrow \text{c[TXTFIRST]}; \\ 3 : \text{s.txt} \rightarrow \text{val c.txt}; \\ 4 : \text{c.display(c.txt)}; \\ 5 : \text{ks.getKey()} \rightarrow \text{val s.key}; \\ 6 : \text{s.key} \rightarrow \text{val c.key}; \\ 7 : \text{c.decrypt(c.key)} \end{array} \right) +_{\text{s}} \left(\begin{array}{l} 8 : \text{ks.getKey()} \rightarrow \text{val s.key}; \\ 9 : \text{s} \rightarrow \text{c[KEYFIRST]}; \\ 10 : \text{s.key} \rightarrow \text{val c.key}; \\ 11 : \text{c.decrypt(c.key)}; \\ 12 : \text{cs.getText()} \rightarrow \text{val s.txt}; \\ 13 : \text{s.txt} \rightarrow \text{val c.txt}; \\ 13 : \text{c.display(c.txt)} \end{array} \right)$$

Compared to O_3 , these implementations are much larger because they require programmers to statically encode all desired schedules. One can easily forget to include some schedules or encode them incorrectly: for instance, if one moved line 12 up to line 10 above, it would eliminate the extra concurrency that was gained by receiving the messages out of order. Thus, our approach is more robust and simpler for the programmer.

On the other hand, nondeterministic choreographies can express some programs that our model cannot. For example, choreographic choice can assign different variable names to messages, according to their arrival order. Other choreographic languages include nondeterministic operators [22, 6], but they do not support computation (a requirement for choreographic programming) or recursion.

Choral is arguably the most powerful implementation of choreographic programming to date, but there are also others that target, e.g., Haskell, Java, Jolie, and Rust [8, 9, 29, 32, 21]. We believe that implementing the out-of-order semantics of O_3 in these languages is possible, too. However, it would likely require more work that touches also the implementation of EPP because, differently from Choral, these languages do not support user-defined communication primitives. Since Choral is more expressive than all other current choreographic programming languages, we have targeted the most general case.

In [23], the authors introduce a Choral library for handling protocols that might deliver messages out of order. Unlike Ozone, this library requires explicitly writing which parts of a choreography are independent. Dependencies between actions also need to be managed at a low level via side-effects. In our approach, out-of-order communications can be elegantly combined by using futures. Furthermore, the work in [23] does not deal with CIVs (which might arise if programmers are not careful) and presents no formal model.

A more loosely related line of research is that on multiparty session types (MPSTs) [19], where abstract choreographies without data or computation are used as protocol specifications that are compiled to “local session types”. Similarly to most work on choreographic programming, some works on MPSTs allow for non-blocking send, but not non-blocking receive as in O_3 . Previous work considered reordering actions in local session types [14], but these reorderings are necessarily limited because asynchronous multiparty session subtyping is undecidable in general [5]. Interprocedural MPSTs have been presented [15], but unlike O_3 , the procedure calls require a central coordinator. Similar comments hold for recent investigations that add nondeterminism because of crashes [1, 33]. More generally, it is unclear whether “concurrency up to data dependency” could be expressed with MPSTs in their current form, since the types do not encode data dependencies.

7 Conclusion

We investigated a model for choreographic programming in which processes can execute out of order and messages can be reordered by the network. These features improve the performance of choreographies, without requiring programmers to rewrite their code, by allowing processes to better overlap communication with computation. However, compilers that use these features must have mechanisms in place to prevent communication integrity violations (CIVs). We presented a scheme to prevent CIVs by attaching dynamically-computed integrity keys to each message. Our results enlarge the class of behaviors that can be captured with choreographic programming without renouncing its correctness guarantees.

An important subject for future work is confluence. Statements can read and write to the local state of a process, so executing statements out of order can cause nondeterminism. Sometimes this nondeterminism is desirable (for instance, to implement consensus algorithms) but sometimes the nondeterminism is unexpected and causes bugs. In our formal model, nondeterminism could be controlled manually by allowing programmers to insert synthetic data dependencies. For example, below we use a hypothetical keyword barrier_p to prevent a file from being closed before it has been written-to:

```
val p.file = open("foo.txt"); p.write(p.file, "hello"); barrier_p; p.close(p.file)
```

More generally, future work could develop a static analysis that identifies when two statements are not safe to execute out of order.

Another opportunity for static analysis to improve on our work concerns the size of session tokens. We chose to represent session tokens as lists of integers, which allowed processes to compute new session tokens without coordinating with one another. However, this encoding means the size of a token is proportional to the depth of the call stack – a problem for tail-recursive programs such as *StreamIt* in Figure 7b. Fortunately, it is easy to see that communication integrity in *StreamIt* could be achieved in constant space by representing the token as a single integer, incremented upon each recursive call, assuming that processes do not participate in multiple instances of the choreography concurrently. With static analysis, a compiler could identify such programs and use a more efficient session token representation.

References

- 1 Manuel Adameit, Kirstin Peters, and Uwe Nestmann. Session types for link failures. In Ahmed Bouajjani and Alexandra Silva, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 37th IFIP WG 6.1 International Conference, FORTE 2017, Held as Part of the 12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017, Neuchâtel, Switzerland, June 19-22, 2017, Proceedings*, volume 10321 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2017. doi:10.1007/978-3-319-60225-7_1.
- 2 Gul Agha. *ACTORS - a Model of Concurrent Computation in Distributed Systems*. MIT Press Series in Artificial Intelligence. MIT Press, Cambridge, MA, 1990.
- 3 Akka. <https://akka.io/>, 2024.
- 4 Henry C. Baker and Carl Hewitt. The incremental garbage collection of processes. In James Low, editor, *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages, USA, August 15-17, 1977*, pages 55–59. ACM, 1977. doi:10.1145/800228.806932.
- 5 Mario Bravetti, Marco Carbone, Julien Lange, Nobuko Yoshida, and Gianluigi Zavattaro. A sound algorithm for asynchronous session subtyping and its implementation. *Log. Methods Comput. Sci.*, 17(1), 2021. URL: <https://lmcs.episciences.org/7238>.
- 6 Mario Bravetti, Ivan Lanese, and Gianluigi Zavattaro. Contract-driven implementation of choreographies. In Christos Kaklamanis and Flemming Nielson, editors, *Trustworthy Global Computing, 4th International Symposium, TGC 2008, Barcelona, Spain, November 3-4, 2008, Revised Selected Papers*, volume 5474 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2008. doi:10.1007/978-3-642-00945-7_1.
- 7 Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured Communication-Centered Programming for Web Services. *ACM Transactions on Programming Languages and Systems*, 34(2):1–78, June 2012. doi:10.1145/2220365.2220367.
- 8 Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: Multiparty asynchronous global programming. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 263–274. ACM, 2013. doi:10.1145/2429069.2429101.
- 9 Luís Cruz-Filipe, Anne Madsen, Fabrizio Montesi, and Marco Peressotti. Modular choreographies: Bridging alice and bob notation to java. In Gokila Dorai, Maurizio Gabbielli, Giulio Manzonetto, Aomar Osmani, Marco Prandini, Gianluigi Zavattaro, and Olaf Zimmermann, editors, *Joint Post-proceedings of the Third and Fourth International Conference on Microservices, Microservices 2020/2022, May 10-12, 2022, Paris, France*, volume 111 of *OASIcs*, pages 3:1–3:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/OASIcs.MICROSERVICES.2020–2022.3.
- 10 Luís Cruz-Filipe and Fabrizio Montesi. On Asynchrony and Choreographies. *Electronic Proceedings in Theoretical Computer Science*, 261:76–90, November 2017. doi:10.4204/EPTCS.261.8.

- 11 Luís Cruz-Filipe and Fabrizio Montesi. Procedural choreographic programming. In Ahmed Bouajjani and Alexandra Silva, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 37th IFIP WG 6.1 International Conference, FORTE 2017, Held as Part of the 12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017, Neuchâtel, Switzerland, June 19-22, 2017, Proceedings*, volume 10321 of *Lecture Notes in Computer Science*, pages 92–107. Springer, 2017. doi:[10.1007/978-3-319-60225-7_7](https://doi.org/10.1007/978-3-319-60225-7_7).
- 12 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Communications in choreographies, revisited. 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 1248–1255. ACM, 2018. doi:[10.1145/3167132.3167267](https://doi.org/10.1145/3167132.3167267).
- 13 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. A formal theory of choreographic programming. *J. Autom. Reason.*, 67(2):21, 2023. doi:[10.1007/S10817-023-09665-3](https://doi.org/10.1007/S10817-023-09665-3).
- 14 Zak Cutner, Nobuko Yoshida, and Martin Vassor. Deadlock-free asynchronous message reordering in rust with multiparty session types. In Jaejin Lee, Kunal Agrawal, and Michael F. Spear, editors, *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, pages 246–261. ACM, 2022. doi:[10.1145/3503221.3508404](https://doi.org/10.1145/3503221.3508404).
- 15 Romain Demangeon and Kohei Honda. Nested protocols in session types. In Maciej Koutny and Irek Ulidowski, editors, *CONCUR 2012 - Concurrency Theory - 23rd International Conference, CONCUR 2012, Newcastle upon Tyne, UK, September 4-7, 2012. Proceedings*, volume 7454 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2012. doi:[10.1007/978-3-642-32940-1_20](https://doi.org/10.1007/978-3-642-32940-1_20).
- 16 Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. Choral: Object-oriented choreographic programming. *ACM Trans. Program. Lang. Syst.*, 46(1):1:1–1:59, 2024. doi:[10.1145/3632398](https://doi.org/10.1145/3632398).
- 17 Eva Graversen, Andrew K. Hirsch, and Fabrizio Montesi. Alice or bob?: Process polymorphism in choreographies. *J. Funct. Program.*, 34, 2024. doi:[10.1017/S0956796823000114](https://doi.org/10.1017/S0956796823000114).
- 18 Andrew K. Hirsch and Deepak Garg. Pirouette: Higher-order typed functional choreographies. *Proc. ACM Program. Lang.*, 6(POPL):1–27, 2022. doi:[10.1145/3498684](https://doi.org/10.1145/3498684).
- 19 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016. doi:[10.1145/2827695](https://doi.org/10.1145/2827695).
- 20 Sung-Shik Jongmans and Petra van den Bos. A predicate transformer for choreographies - computing preconditions in choreographic programming. In Ilya Sergey, editor, *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, volume 13240 of *Lecture Notes in Computer Science*, pages 520–547. Springer, 2022. doi:[10.1007/978-3-030-99336-8_19](https://doi.org/10.1007/978-3-030-99336-8_19).
- 21 Shun Kashiwa, Gan Shen, Soroush Zare, and Lindsey Kuper. Portable, efficient, and practical library-level choreographic programming. *CoRR*, abs/2311.11472, 2023. doi:[10.48550/arXiv.2311.11472](https://doi.org/10.48550/arXiv.2311.11472).
- 22 Ivan Lanese, Claudio Guidi, Fabrizio Montesi, and Gianluigi Zavattaro. Bridging the gap between interaction- and process-oriented choreographies. In Antonio Cerone and Stefan Gruner, editors, *Sixth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2008, Cape Town, South Africa, 10-14 November 2008*, pages 323–332. IEEE Computer Society, 2008. doi:[10.1109/SEFM.2008.11](https://doi.org/10.1109/SEFM.2008.11).
- 23 Lovro Lugović and Fabrizio Montesi. Real-world choreographic programming: Full-duplex asynchrony and interoperability. *The Art, Science, and Engineering of Programming*, 8(2), October 2023. doi:[10.22152/programming-journal.org/2024/8/8](https://doi.org/10.22152/programming-journal.org/2024/8/8).
- 24 Massimo Merro and Davide Sangiorgi. On asynchrony in name-passing calculi. *Mathematical Structures in Computer Science*, 14(5):715–767, October 2004. doi:[10.1017/S0960129504004323](https://doi.org/10.1017/S0960129504004323).

- 25 Fabrizio Montesi. *Choreographic Programming*. Ph.D. thesis, IT University of Copenhagen, 2013. URL: <https://www.fabriziomontesi.com/files/choreographic-programming.pdf>.
- 26 Fabrizio Montesi. *Introduction to Choreographies*. Cambridge University Press, Cambridge, 2023.
- 27 Dan Plyukhin, Marco Peressotti, and Fabrizio Montesi. Ozone: Fully out-of-order choreographies. *CoRR*, abs/2401.17403, 2024. doi:10.48550/arXiv.2401.17403.
- 28 Johannes Aman Pohjola, Alejandro Gómez-Londoño, James Shaker, and Michael Norrish. Kalas: A Verified, End-To-End Compiler for a Choreographic Language. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel*, volume 237 of *LIPICS*, pages 27:1–27:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.ITP.2022.27.
- 29 Mila Dalla Preda, Maurizio Gabbielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic choreographies: Theory and implementation. *Log. Methods Comput. Sci.*, 13(2), 2017. doi:10.23638/LMCS-13(2:1)2017.
- 30 Zongyan Qiu, Xiangpeng Zhao, Chao Cai, and Hongli Yang. Towards the theoretical foundation of choreography. In *Proceedings of the 16th International Conference on World Wide Web*, pages 973–982, Banff Alberta Canada, May 2007. ACM. doi:10.1145/1242572.1242704.
- 31 Michael Scharf and Sebastian Kiesel. Head-of-line Blocking in TCP and SCTP: Analysis and Measurements. In *Proceedings of the Global Telecommunications Conference, 2006. GLOBECOM '06, San Francisco, CA, USA, 27 November - 1 December 2006*. IEEE, 2006. doi:10.1109/GLOCOM.2006.333.
- 32 Gan Shen, Shun Kashiwa, and Lindsey Kuper. Haschor: Functional choreographic programming for all (functional pearl). *Proc. ACM Program. Lang.*, 7(ICFP):541–565, 2023. doi:10.1145/3607849.
- 33 Malte Viering, Raymond Hu, Patrick Eugster, and Lukasz Ziarek. A multiparty session typing discipline for fault-tolerant event-driven distributed programming. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–30, 2021. doi:10.1145/3485501.
- 34 Stephanie Wang, Eric Liang, Edward Oakes, Benjamin Hindman, Frank Sifei Luan, Audrey Cheng, and Ion Stoica. Ownership: A distributed futures system for fine-grained tasks. In James Mickens and Renata Teixeira, editors, *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*, pages 671–686. USENIX Association, 2021. URL: <https://www.usenix.org/conference/nsdi21/presentation/cheng>.

Tenspiler: A Verified-Lifting-Based Compiler for Tensor Operations

Jie Qiu 

Pittsburgh, PA, USA

Colin Cai 

University of California, Berkeley, CA, USA

Sahil Bhatia 

University of California, Berkeley, CA, USA

Niranjan Hasabnis 

Intel Labs, Menlo Park, CA, USA

Sanjit A. Seshia 

University of California, Berkeley, CA, USA

Alvin Cheung 

University of California, Berkeley, CA, USA

Abstract

Tensor processing infrastructures such as deep learning frameworks and specialized hardware accelerators have revolutionized how computationally intensive code from domains such as deep learning and image processing is executed and optimized. These infrastructures provide powerful and expressive abstractions while ensuring high performance. However, to utilize them, code must be written specifically using the APIs / ISAs of such software frameworks or hardware accelerators. Importantly, given the fast pace of innovation in these domains, code written today quickly becomes legacy as new frameworks and accelerators are developed, and migrating such legacy code manually is a considerable effort.

To enable developers in leveraging such DSLs while preserving their current programming paradigm, we present TENSPILER, a verified-lifting-based compiler that uses program synthesis to translate sequential programs written in general-purpose programming languages (e.g., C++ or Python code that does not leverage any specialized framework or accelerator) into tensor operations. Central to TENSPILER is our carefully crafted yet simple intermediate language, named TENSIR, that expresses tensor operations. TENSIR enables efficient lifting, verification, and code generation. Unlike classical pattern-matching-based compilers, TENSPILER uses program synthesis to translate input code into TENSIR, which is then compiled to the target API / ISA. Currently, TENSPILER already supports **six** DSLs, spanning a broad spectrum of software and hardware environments. Furthermore, we show that new backends can be easily supported by TENSPILER by adding simple pattern-matching rules for TENSIR. Using 10 real-world code benchmark suites, our experimental evaluation shows that by translating code to be executed on 6 different software frameworks and hardware devices, TENSPILER offers on average **105 \times** kernel and **9.65 \times** end-to-end execution time improvement over the fully-optimized sequential implementation of the same benchmarks.

2012 ACM Subject Classification Software and its engineering → Compilers

Keywords and phrases Program Synthesis, Code Transpilation, Tensor DSLs, Verification

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.32

Related Version Full Version: <https://arxiv.org/abs/2404.18249> [29]

Supplementary Material Software (ECOOP 2024 Artifact Evaluation approved artifact):
<https://doi.org/10.4230/DARTS.10.2.17>



© Jie Qiu, Colin Cai, Sahil Bhatia, Niranjan Hasabnis, Sanjit A. Seshia, and Alvin Cheung;

licensed under Creative Commons License CC-BY 4.0

38th European Conference on Object-Oriented Programming (ECOOP 2024).

Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 32; pp. 32:1–32:28



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

Funding This work was supported in part by DARPA Contract FA8750-23-C-0080, a Google BAIR Commons project, NSF grants IIS-1955488, IIS-2027575, ARO W911NF2110339, ONR N00014-21-1-2724, and DOE award DE-SC0016260, DE-SC0021982, and the Sloan Foundation.

Acknowledgements We would like to thank Jayaram Bobba and Zhongkai Zhang from Intel’s Habana team for inputs on Gaudi architecture, TPC-C programming model, and obtaining high-performance from TPC kernels. We would like to thank Hasan Genc and Sophia Shao for helpful insights into Gemmini code generation.

1 Introduction

We have witnessed an explosion of new computational infrastructures for tensor computation in recent years: from software frameworks such as TensorFlow to specialized hardware accelerators like tensor processing units. Such infrastructures arise due to new application domains such as image processing and training deep learning (DL) models, and they often expose their functionality via various domain-specific languages (DSLs) that range from specialized instruction sets such as vectorized instructions to high-level programming interfaces such as Apple’s MLX [27] or TensorFlow’s XLA [37].

To leverage the optimization offered by such infrastructures, applications must be written against the provided programming interfaces: developers must first master each DSL’s programming model to write new code, and existing applications must be rewritten. This problem is recurring as new DSLs keep appearing targeting different application domains. Manually rewriting existing applications is tedious and increases the likelihood of introducing bugs. The classical way of addressing such issues is to build transpilers [15, 5, 6, 19, 25, 26] that translate code from paradigms developers are familiar with (e.g., C++ code using the STL library) to the one provided by the target DSL (e.g., NumPy API). Nonetheless, building such a transpiler is resource-intensive, error-prone, and each one is specialized to a specific target DSL. For instance, existing compilers such as Dexter [6], STNG [19], and C2TACO [25] target specific DSLs like Halide and TACO, and are not easily extensible to support new operators or backends. While recently developed DL models such as GPT have shown promise in code translation, they do not provide any guarantees on the correctness of output. Moreover, GPT fails to generate even syntactically correct code for DSLs it has not seen in training data, limiting its applicability to new or less popular DSLs.

In this paper, we describe a tensor compiler that addresses these challenges. We introduce TENSPILER— a compiler designed to automate the transpilation of code to *multiple* tensor processing frameworks and hardware accelerators. TENSPILER uses verified lifting [12] (VL), a technique using inductive program synthesis to infer provably equivalent program summaries expressed using a user-defined intermediate representation (IR), and generate executable code from the synthesized summary to the target DSL. In contrast to conventional compilers that rely on pattern-matching to compile code, VL uses a search-based technique for the translation process. The two key steps of VL are:

- **Search Phase:** This stage lifts the input code to an equivalent program written using a user-provided IR, where the IR is used to model the functionality of each operator in the target DSL. Lifting is formulated as a syntax-guided synthesis [8] problem.
- **Verification:** Once lifted, a theorem prover is used to validate if the synthesized summary is functionally equivalent to the input. If so, executable code is produced by calling the user-provided code generator from the summary; otherwise, another summary is generated by the search phase.

The key to making lifting efficient lies in the design of the IR (i.e., how the target DSL is modeled). In prior work [5, 6, 7, 34, 21], each function or instruction exposed by the target DSL is modeled explicitly. While doing so makes the search efficient, such explicit modeling makes the compiler hard to extend to other DSLs. With TENSPILER, we introduce, for the first time, a *single unified* IR, TENSIR, that is designed for tensor operations and can easily generate code to *multiple* tensor processing software frameworks and hardware accelerators. Surprisingly, TENSIR is a small language based on tensor algebra that includes commonly used vector and matrix operations. While other unifying IR exists (e.g., MLIR [23]), they are targeted for classical pattern-matching compilers. As we will discuss in Sec. 5 and Sec. 4.2, TENSIR instead is designed for synthesis-based compilers and thus aims to enable both efficient search and verification.

In summary, this paper makes the following contributions:

1. We describe the design of TENSIR for transpiling code to tensor processing DSLs. TENSIR is simple yet expressive enough to model the functionalities provided by different software frameworks and hardware accelerators, and enables efficient code transpilation using verified lifting, as detailed in Sec. 3.
2. Based on TENSIR, we devise various optimization techniques to make synthesis and verification tractable, and scale to real-world programs in Sec. 5.
3. We implement TENSPILER, a verified lifting-driven transpiler built using TENSIR as the modeling language. We demonstrate the effectiveness of TENSPILER by using it to lift real-world code from 10 different suites to **6** different open-source and commercially-available tensor processing software frameworks and hardware accelerators. We illustrate the ease of constructing such transpilers by building one for MLX, a new tensor processing framework that was released only four months ago, using less than **200** lines of code in Sec. 6.

We have released Tenspiler’s code on <https://github.com/tenspiler/tenspiler>.

2 Overview

TENSPILER takes in C/C++ or Python code as input¹ and transpiles it to a functionally equivalent program that leverages different software frameworks and hardware accelerators (details described in Sec. 4.3) for tensor computation. As mentioned, TENSPILER uses verified lifting to first translate the source program into TENSIR. Unlike traditional pattern-matching compilers, TENSPILER formulates code translation as a search for a program expressed in TENSIR that is provably semantic-equivalent to the input. Doing so avoids the need to devise pattern-matching rules and prove their correctness. To make the search scalable, instead of directly searching within the DSL exposed by each target, we designed a high-level IR called TENSIR that abstracts away the low-level implementation details of each DSL operator and captures only their semantics, unifying various DSLs into a common set of tensor operators. TENSPILER uses a program synthesizer (currently Rosette [36], a synthesizer for finite domain theories) to lift the input code to TENSIR during the search phase. The synthesized output is then verified using a theorem prover (currently, an SMT solver, CVC5 [10]) for the unbounded domain. “Unbounded domain” means the verification

¹ TENSPILER currently supports a subset of the C/C++ and Python language (in particular it does not support code that uses pointers or objects, which we have not encountered such use in our benchmarks). It also expects any external libraries used in the input to be functionally modeled, which is how TENSPILER currently supports code that uses the `STL::vector` library.

32:4 Tenspiler: A Verified-Lifting-Based Compiler for Tensor Operations

```

1  inline uint8_t screen_8x8 (uint8_t a, uint8_t b) { return a + b - (a * b) / 255; }
2  vector<vector<int>> screen_blend(vector<vector<int>> b, vector<vector<int>> a) {
3    vector<vector<int>> out; int m = b.size(); int n = b[0].size();
4    for (int row = 0; row < m; row++) {
5      vector<int> r_v;
6      for (int col = 0; col < n; col++)
7        r_v.push_back(screen_8x8(b[col, row], a(col, row)));
8      out.push_back(r_v);
9    return out;

```

(a) Original Blend function in C++.

```

1  def t_t(x, y, operation):
2    if len(x) < 1 or len(x) != len(y): return []
3    else: return [operation(x[0], y[0])] + t_t(x[1:], y[1:], operation)
4
5  def t_s(x, a, operation):
6    if len(x) < 1: return []
7    else: return [operation(x[0]), a] + t_s(x[1:], a, operation)

```

(b) Operators in TENSIR. We represent tensor_scalar as t_s and tensor_tensor as t_t.

```

1  def inner_loop(row, col, b, a, r_v, out):
2    return col >= 0 and col <= len(b[0]) and row >= 0 and row < len(b) and
3    r_v == t_t(t_t(b[row][:col], a[row][:col], +),
4                t_s(t_t(b[row][:col], a[row][:col], *), 255, /), -) and
5    out == t_t(t_t(b[:row], a[:row], +), t_s(t_t(b[:row], a[:row], *), 255, /), -)
6
7  def outer_loop(row, col, b, a, row_vec, out):
8    return row >= 0 and row < len(b) and
9    out == t_t(t_t(b[:row], a[:row], +), t_s(t_t(b[:row], a[:row], *), 255, /), -)

```

(c) Synthesized loop invariants.

```

1  def screen_blend(b, a): return b + a - b * a // 255 # NumPy/TensorFlow/PyTorch/MLX
2  uchar256 Screen8x8(uchar256 a, uchar256 b) { # TPC-C implementation for Gaudi
3    uchar256 c = v_u8_mul_b(a, b) * v_reciprocal_fast_f32(255);
4    uchar256 d = v_u8_add_b(a, v_u8_sub_b(b, c));
5    return d; }

```

(d) Generated executable code for different tensor processing DSL.

Figure 1 End-to-End example of using TENSPILER to transpile code.

is performed for all possible program states, not just a bounded set of states (e.g., all states where integers are represented using 8 bits) that Rosette considers during the synthesis phase. Once verified, TENSPILER then translates the TENSIR program to the concrete syntax of the target DSL using simple pattern-matching rules.

We illustrate TENSPILER using the example in Fig. 1a as our **S** (source), where **S** implements blending, a common image processing operation. It lightens the base color by iterating over each pixel, implemented as a nested loop over all the **rows** and **cols** in the image. Our goal is to transpile this code to the target DSLs supported by TENSPILER as shown in Fig. 1d.

TENSPILER first translates the input code to TENSIR. To be discussed in Sec. 4, TENSIR consists of several operators that model common tensor algebra operations, two of which are shown in Fig. 1b. The **t_t** function performs element-wise operations (one of **+**, **-**, *****, **/**, **%**) on tensors **x** and **y** and is defined recursively on each element. Meanwhile, **t_s** performs element-wise scalar operations on tensor **x** using the scalar value **a** and is similarly defined. Importantly, both operators are purely functional models of the tensor operations that lack implementation details that a specific target might leverage (e.g., tiling, vectorization, etc). The idea is that if **S** can be expressed using only these operators via lifting, then the lifted program can be easily translated to the targeted backends.

In TENSPILER, lifting is formulated as a Syntax-Guided Synthesis (SyGuS) [8] problem, where the goal is to synthesize a semantically equivalent program summary (PS), represented as a sequence of operators from our TENSIR, with the input code as the specification. A search space (specified using grammar) describes the set of potential candidate programs for the given specification. An input program S is semantically equivalent to the synthesized expression S' if for all possible program inputs i , $S(i) = S'(i)$.

TENSPILER uses symbolic search to solve the synthesis problem. Symbolic search is typically implemented through enumerative or deductive search, and using constraint-solving approaches which often rely on domain-specific heuristics to scale. As a SyGuS problem, symbolic search is implemented as enumerating different expressions over a user-provided grammar, where the grammar encodes all possible combinations of operators in the target DSL up to a specified depth. However, as the depth increases, the number of choices grows exponentially, making the search intractable. As we will discuss in Sec. 5, TENSIR is designed to make synthesis scalable. For S , the synthesis phase returns the following solution:

```
def lifted_program(b, a): return t_t(t_t(b, a, +), t_s(t_t(b, a, *), 255, /), -)
```

As TENSPILER’s synthesizer currently can only reason about finite domains, all synthesized solutions are checked for full functional equivalence using an automated theorem prover. Since S has loops, checking equivalence with the generated program on all inputs requires loop invariants. Such invariants are synthesized during the synthesis phase by constructing a grammar similar to the PS grammar. For instance, for S , the synthesis phase yields two loop invariants (one for each loop) alongside a PS . As shown in Fig. 1c, these loop invariants are not arbitrary; within the loop invariants, the output variable, `out`, is expressed as a combination of operators from the TENSIR that help prove the synthesized PS . We will leverage this to improve synthesis efficiency, to be explained in Sec. 5.

With the synthesized solution expressed in TENSIR, the final step is to translate it into the concrete syntax of the target DSL(s). In TENSPILER, this is done via simple pattern-matching rules. In Fig. 1d, we present the translated code for different supported DSLs. As we will discuss in Sec. 4.3, TENSIR is designed such that generating executable code is straightforward. In fact, the code generators for the different tensor processing infrastructures supported by TENSPILER are highly similar to each other, as we will discuss in Sec. 6.

3 The TENSIR Intermediate Representation

We now discuss our intermediate representation, TENSIR, which plays a pivotal role in TENSPILER. While prior lifting-based compilers all use search to compile programs, their IRs are specialized for their use cases. Those IRs consist of operators from the target languages, describing their high-level semantics while avoiding low-level implementation details. For example, Casper [5] was built to translate sequential Java to MapReduce programs. The MapReduce framework consists of several versions of `map` that differ by their input types. However, Casper only defines one operator in its IR that models `map`’s functionality, and decides on the implementation to use during code generation. While doing so makes synthesis tractable, it also makes the compiler inflexible as adding another target (e.g., a hardware accelerator that supports `map` over tensors) will require modeling its functionality, which may be incompatible with the existing `map` from Casper’s IR.

To address this challenge, we designed a novel IR, TENSIR, by studying the DSLs provided by various software frameworks and hardware accelerators for tensor computation. TENSIR is rooted in tensor algebra and is designed for flexibility, allowing translation to both software (deep learning frameworks, vector processing libraries) and hardware environments (machine

learning accelerators) as to be discussed in Sec. 4.3. This flexibility enables developers to select which target to execute the translated code based on availability and specific performance requirements. Given the dynamic nature of tensor processing infrastructures, TENSIR can be modified easily in terms of both adding support for new tensor operators and new target backends. This is illustrated in Sec. 6, where it only took **200** lines of code for TENSPILER to support Apple’s recently introduced MLX framework [27].

Comparison with MLIR. MLIR [23] is a compiler infrastructure that enables the representation and transformation of code at various levels of abstraction. The core idea behind MLIR is to provide a unified IR that can capture the semantics of the program at different levels of detail (dialects), from high-level abstractions down to low-level, target-specific instructions. Developers can use MLIR by progressively lowering the code through different dialects until it reaches a level suitable for the target hardware. While MLIR and its dialects offer a powerful infrastructure for progressively targeting multiple hardware backends, we found that the existing dialects do not fully support all the operators required for our use case. Independently, both the `linalg` and `tensor` dialects do not support all the operators TENSIR supports. For example, the `select` operator, which is crucial for image processing kernels that apply operations conditioned on pixel values, is not supported by any MLIR dialects. Additionally, unifying these dialects can be challenging for developers. Instead of unifying, recent work such as mlirSynth [14] has explored using program synthesis to translate between different MLIR dialects. In contrast, TENSIR is designed to be flexible and easily extensible. Developers can add new operators to TENSIR by simply describing their high-level semantics, and new backend support can be incorporated by defining simple pattern-matching rules. This approach allows developers to extend TENSIR without going into the intricacies of MLIR. Moreover, TENSIR can practically be compiled into different MLIR dialects, providing developers the flexibility to leverage the MLIR infrastructure if desired.

3.1 Language Definition

The operators and grammar of TENSIR are shown in Fig. 2. TENSIR operates on tensors² and includes various operations. The core strength of TENSIR lies in `tensorOp`, which forms the backbone of tensor operations, including a diverse range of manipulations on tensors, such as element-wise operations, tensor vector multiplication, and reductions. These are grouped into different categories:

- `tensor_scalar` operations describe element-wise operations involving tensors and scalars, such as scalar multiplication of each element in a matrix.
- `tensor_tensor` operations perform element-wise operations between two tensors, such as element-wise multiplication of two tensors.
- Tensor reshaping such as `transpose`.
- `tensor_vec_prod` operation denotes tensor-vector products, enabling operations like matrix-vector multiplication.³
- Tensor reductions such as `reduce_max` and `reduce_sum`, which focus on aggregating tensor values, with the former determining the maximum element and the latter computing the sum across specified dimensions.

² In the grammar, the *Tensor Literal* refers to 1D or 2D tensors as we did not encounter higher dimensional tensors in our benchmarks.

³ While we can also define a tensor-tensor product operator, we did not encounter such benchmarks in our evaluation and hence omitted it from TENSIR’s grammar.

```

 $p \in Op := s_{comp} \mid t_{comp} \mid c_{control}$ 
 $t_{comp} \in tensorOp := tensor\_scalar(t, l, o) \mid tensor\_tensor(t, t, o) \mid$ 
 $\quad transpose(t) \mid tensor\_vec\_prod(t, t) \mid a$ 
 $s_{comp} \in scalarOp := reduce\_max(t) \mid reduce\_sum(t)$ 
 $o \in op := + \mid - \mid / \mid * \mid \%$ 
 $c_{control} \in controlflowOp := ite(cond, i, i)$ 
 $cond \in boolExpr := i \ rop \ i$ 
 $i \in inp := l \mid t$ 
 $rop \in relOp := > \mid < \mid == \mid \neg$ 
 $a \in accessOp := take(t, l) \mid tail(t, l) \mid slice(t, l, l_1, l_2)$ 
 $l := Integer\ Literal \mid size(t, l) \mid s_{comp}$ 
 $t := Tensor\ Literal \mid t_{comp}$ 

```

■ **Figure 2** TENSIR grammar.

The recursive nature of TENSIR’s grammar allows tensor operations to be composed, facilitating the expression of complex algorithms encountered in source code. TENSIR extends its expressiveness beyond tensor operations by also including a control flow operator (`controlflowOp`). It integrates control flow through the `ite` operator, enabling conditional logic into tensor computation. This operator is crucial for translating real-world loopy programs that contain branches.

TENSIR is notable not only for its diversity of operations but also for the granularity of each operator, which significantly enhances its utility in translating code. The fine-grained nature of operations, from basic element-wise computations to advanced tensor reductions and control flow constructs, allows the grammar to capture the diverse tensor computation present in the input. Moreover, the selected set of operations aligns with the core functionalities supported by most tensor processing infrastructures. This ensures that TENSIR can seamlessly integrate with various frameworks and accelerators, offering flexibility in supporting multiple target DSLs. This comprehensive yet concise grammar serves as a bridge between traditional loop-based programming paradigms and the highly parallelizable world of tensor computation, providing a clear and expressive language for describing mathematical operations on tensors.

Besides tensor operations, TENSIR also supports tensor accessing and manipulation:

- `take(t, n)` extracts `n` elements from the beginning.
- `tail(t, n)` returns all the elements in tensor `t` after the first `n` elements.
- `slice(t, l, s, e)` extracts a contiguous sub-tensor from `t` from indices `s` (inclusive) to `e` (exclusive) along dimension `l`.
- `size(t, l)` returns the size of tensor `t` in dimension `l`.

Such functions are used to express the loop invariants and program summaries in the synthesis phase, as we describe next.

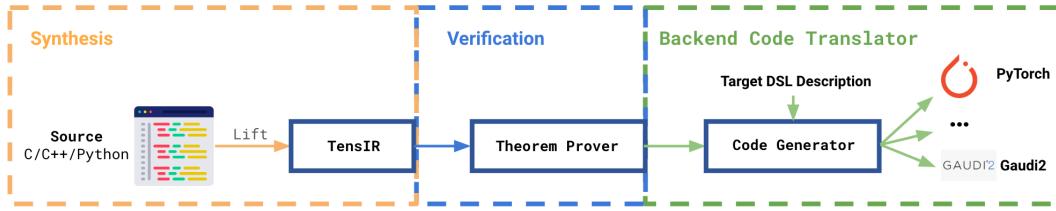


Figure 3 An overview of the TENSPILER Framework.

4 Transpiling Code Using Tenspiler

As shown in Fig. 3, TENSPILER is designed to translate a program in high-level languages, source (S), into another program that leverages different tensor processing infrastructures. TENSPILER currently support a vector processing library (NumPy) for CPU execution, DL frameworks (PyTorch, TensorFlow, MLX) for GPU execution, and ISAs for specialized hardware accelerators (Gaudi, Gemmini). TENSPILER is a verified-lifting-based compiler, leveraging search to find a program within the target domain. Instead of relying on traditional pattern-matching rules, TENSPILER translates source programs with a 3-phase workflow:

1. synthesis,
2. verification, and
3. backend code generation.

TENSPILER uses a single IR, TENSIR, to facilitate all 3 phases of its workflow. TENSIR is designed to include tensor processing operators common to all target backends. As shown in the figure, the synthesis phase takes in S and generates a program summary expressed using TENSIR. Then, in the verification phase, TENSPILER verifies the generated summaries to ensure their semantic equivalence with S . Finally, in the code generation phase, the TENSIR program is translated to the concrete syntax of the target DSL(s).

4.1 Synthesis

The objective of this phase is to search for a program expressed using the operators in TENSIR, and to ensure that the generated program is semantically equivalent to S . We formulate the search as a SyGuS problem [8] characterized by three parameters:

1. the **specification** describing the property the synthesized TENSIR expression should satisfy,
2. the **search space** that describes the space of possible solutions,
3. the **search algorithm** which searches for the candidate programs.

For TENSPILER, the specification is to find a functionally equivalent program to S . Various methods exist to express this specification, such as using input-output examples, bounded-model checking, and verification conditions (VC) [18].

In TENSPILER, we use VCs as the specification for the synthesis phase as it provides full guarantees (i.e., for all program states up to a bound, e.g., states where all integers are encoded using 8 bits) on the equivalence of S and the translated program. VCs are logical expressions encoding the correctness properties of S .

Specifically, given a program P with $vars$ representing all the variables appearing in P , and pre , $post$, inv representing the pre-conditions, the post-condition, and the invariant(s), respectively, the VCs for a program with loops consist of the following clauses:

1. **Initial Condition:** $\forall vars. pre(vars) \rightarrow inv(vars)$: loop invariants must hold before the loop begins its execution.
2. **Loop Preservation:** $\forall vars, vars'. inv(vars) \wedge P(vars, vars') \rightarrow inv(vars')$: if the invariant holds before a loop iteration, they should continue to hold after that iteration.
3. **Post-Condition:** $\forall vars. inv(vars) \rightarrow post(vars)$: invariants should hold once the loop has completed its execution.

There exist standard techniques for generating VCs from a given source program [11]. In TENSPILER, the *PS* and invariants in the VCs are generated as placeholders as **S** is analyzed, with their bodies to be synthesized during the synthesis phase.

Next, we define the search space for synthesis. This space outlines the potential solutions for both the *PS* and invariant(s), describing the solutions that could potentially satisfy the VC. Expressed as a context-free grammar (CFG), the search space imposes syntactic constraints on the structure of the outputs. In TENSPILER, the goal is not to find any *PS* or invariants but ones that represent the output variables in **S** as some sequence of operators from TENSIR, expressed mathematically as:

$$\forall o \in outputVars. o = p, \text{ where } p \in Op \text{ as defined in Fig. 2.} \quad (1)$$

This states that all return variables in **S** should be expressed as a program from TENSIR. With the specification in the form of VCs, *p* expressed using TENSIR, and the search space for the *PS* and invariants, the synthesis problem can be formally defined as:

$$\exists inv_0, inv_1, \dots, PS \in G. \forall \sigma. VC(S, inv_0, inv_1, \dots, PS, \sigma) \quad (2)$$

The goal of synthesis is then to find expressions from the search space *G* for *PS* and *invs* such that, for all program states σ , they satisfy the VC.

For TENSPILER, we use an off-the-shelf symbolic search engine, Rosette [36], which uses constraint solving to address the synthesis problem. In a constraint-solving-based approach, the specification ϕ (i.e., VC) and the search space *G* are encoded as a single formula, and an SMT solver is then utilized to find a model that satisfies the formula. As a constraint-based solver, increasing the number of constraints makes the problem more challenging. Given that ϕ is fixed for a particular benchmark, the design of the *G* becomes crucial. In Sec. 5, we discuss how the design of TENSIR helps keep the grammar size reasonable and scales the synthesis process.

4.2 Verification

During the synthesis phase, as the *PS* and loop invariant(s) are validated against the VC only for a bounded set of program states,⁴ it is essential to check their validity for all program states. TENSPILER uses an SMT solver to do so by negating the VC in program verification i.e., checking if $\neg VC(S, inv_1, inv_2, \dots, PS, \sigma)$ is satisfiable for some σ . The placeholders in the VC are substituted with the synthesized bodies of *PS* and *invs*. If the solver cannot find any such σ , then the generated *PS* and *invs* are correct for all possible program states, thus proving *PS* and *invs* hold for all program states. If a σ is found, then TENSPILER will iterate back to the synthesis phase in search of another candidate expression.

Besides using SMT solvers for Eq. (2), TENSPILER also leverages SMT solvers' support of algebraic data types (ADT) to allow users to define common data structures such as lists and tuples. Internally, TENSPILER models tensors using the list data structure defined using

⁴ We are unaware of any SyGuS solvers that can validate against an unbounded set of program states efficiently, including state of the art solvers such as Z3 and CVC5.

```

1  (assert (forall ((data (Tensor Int)) (a Int) (idx Int))
2  (=> (>= index 0) ∧ (≤= index len(data))
3  (= t_s(data[:idx], a, *) (+ [data[0]*a] t_s(data[1:idx], a, *))))))

```

 **Figure 4** Example of an inductive axiom for the `tensor_scalar` operator in TENSIR described using SMT-LIB. “+” corresponds to the concat operator.

ADTs. We use ADT’s accessor and constructor functions to retrieve and create new tensors. All the tensor accessing functions like `slice`, `take` are modeled as recursive functions over the list data structure. Currently, while image processing kernels use integers and deep learning kernels operate over floats, we verify all the benchmarks using the theory of integers and reals, due to poor solver support for reasoning about floats.

Since the verification of loop invariants is undecidable in general, we define additional *axioms* for the operators in TENSIR to aid verification. These axioms describe the behavior of functions that cannot be automatically deduced by the solver. Identifying the axioms requires an understanding of the program’s semantics and the properties that need verification. Such axioms describe simple attributes such as distributivity, associativity, and commutativity of the tensor operators. In Fig. 4, we show an inductive axiom for the `tensor_scalar` operator which states that the result for a given index is determined by the product of the first element of the tensor and an integer, plus the result for the remaining sub-tensor up to that index. As shown, having tensors as first-class objects in TENSIR greatly simplifies the task of defining these properties. Instead of defining these properties using low-level SMT-LIB list data structures, TENSIR enables users to define them at the tensor level, abstracting away the low-level solver-related details. This high-level representation greatly simplifies the task of defining these properties and makes the axioms more readable and maintainable.

4.3 Code Generation

After successfully verifying the synthesized TENSIR program, the final stage in TENSPILER’s workflow is to translate the TENSIR program into the concrete syntax of a target DSL. TENSIR makes this easy as it inherently represents tensor operations supported by all the target DSLs, and code can be generated using simple syntax-driven rules that map TENSIR operators to their DSL-specific counterparts.

To translate the TENSIR expression into an executable DSL program, our code generation step recursively processes each part of the TENSIR expression. Fig. 5 illustrates a portion of the code generation function for PyTorch. The function maps TENSIR variables to their names (line 3), literals to their values (line 5), and function calls to their PyTorch equivalents based on function signatures (lines 6-15).

Consider the running example in Fig. 1a, where the synthesized TENSIR solution is `t_t(t_t(b, a, +), t_s(t_t(b, a, *), 255, /), -)`. This expression represents a `t_t` function call with the `-` operator, which maps to `torch.subtract` as shown in line 13. Next, the `codegen` function is called recursively on the two arguments, `t_t(b, a, +)` and `t_s(t_t(b, a, *), 255, /)`. This results in the final translated PyTorch expression as `torch.subtract(torch.add(b, a), torch.divide(torch.add(b, a), 255))`.

To extend support for a new backend, one simply needs to replace the DSL operator names in lines 11, 15, 17, and 19. For example, in MLX’s codegen, `torch.add` on line 11 would be replaced by `mlx.core.add`.

This direct and syntactic translation simplifies the integration of new tensor-based target DSL into TENSPILER, as one would only need to add simple translation rules in the code generation process. For instance, we add support for MLX by changing only 65 lines of code to an existing 200-line template, as its API closely follows that of NumPy.

```

1 def codegen(expr: Expr):
2     if isinstance(expr, Var):
3         return expr.name()
4     elif isinstance(expr, Lit):
5         return expr.val()
6     elif isinstance(expr, Call):
7         f_name, args = expr.name(), expr.arguments()
8         if f_name in {"t_t", "t_s"}:
9             op = args[-1]
10            if op == "+":
11                return f"torch.add({codegen(args[0])},{codegen(args[1])})"
12                #corresponding MLX return statement
13                #return f"mlx.core.add({codegen(args[0])},{codegen(args[1])})"
14            elif op == "-":
15                return f"torch.subtract({codegen(args[0])},{codegen(args[1])})"
16            elif op == "*":
17                return f"torch.multiply({codegen(args[0])},{codegen(args[1])})"
18            elif op == "/":
19                return f"torch.divide({codegen(args[0])},{codegen(args[1])})"
20            ...

```

Figure 5 Code generation for the element-wise add operator to different targets.

As a part of this work, we have implemented support for **six** different target DSLs in our code generator: **NumPy**, **TensorFlow**, **PyTorch**, **MLX** (an ML framework for Apple silicon), **TPC-C** (C-based programming language for Intel’s Gaudi processor), and **Gemmini** (an open-source neural network accelerator generator).⁵

5 Synthesis Optimizations

A naive approach to constructing the grammar for search space is to enumerate all possible combinations of TENSIR expressions up to a fixed depth. For TENSIR as defined in Fig. 2, if we focus solely on the compute operators, a depth-4 grammar (i.e., sequence of 4 operators) results in a search space of $\sim 200k$ expressions, since it grows exponentially with the depth and the number of operations. In Fig. 6, we show a small part of the depth-4 grammar. We have devised several optimizations to reduce the search space and make the search tractable.

5.1 Restricting Operators

First, we generate the grammar based on types, i.e., we only include the operators whose output types match with the expected return type. In the case of **S** in Fig. 1a, since the return type is `vector<vector<int>>`, all reduction operations are excluded. In Fig. 6, the operators in v_4 and l_4 will be removed (shown in red). These correspond to operators returning 1-D vectors and integers respectively.

5.2 Restricting Program States

We further optimize the search space by restricting the set of program states in Eq. (2). Instead of satisfying the VC for all σ , we find PS and $invs$ that satisfy a bounded set. Bounded synthesis is crucial because most SyGuS solvers have limited support for recursive function definitions and require SMT solvers for validation. However, SMT solvers lack inherent support for reasoning about TENSIR operators that are not covered by the standard theories defined in SMT-LIB [9] and require additional axioms to be defined. We instead

⁵ We provide further details of these DSLs in Appendix A in the extended version of this paper[29].

```

out   := m4 | v4 | l4
m4  := tensor_scalar(m3, l4, o) | tensor_tensor(m3, m3, o) |
          transpose(m3) | ite(cond4, m3, m3) | m3
v4  := tensor_scalar(v3, l4, o) | tensor_tensor(v3, v3, o) |
          tensor_vec_prod(m3, v3) | ite(cond4, v3, v3) | v3
l4  := reduce_max(l3) | reduce_sum(l3) | l3
cond4 := l3 rop l3
...
l1  := 255 | size(t1)
t1  := a⟨a1, a2 ... an⟩ | b⟨b1, b2 ... bn⟩ a⟨a1, a2⟩ | b⟨b1, b2⟩
rop  := > | < | == | ∼
o    := + | - | / | *

```

 **Figure 6** A depth 4 general synthesis grammar for the source in Fig. 1a.

integrate bounded synthesis by restricting the maximum unrollings of recursive operators, thereby eliminating the need for additional axioms. Specifically, we restrict the program states by limiting the lengths of the data structures and the sizes of the data types. For instance, in TENSPILER, we constrain all 1D tensors to length 2 and the integers to 6 bits or less for the first rounds of synthesis. In Fig. 6, all the tensor literals in t_1 are changed from an unbounded length “n” (shown in orange) to length 2 (shown in blue). If the synthesized choices fail to verify, we then increase the bounds in subsequent rounds. Note that since the synthesized solutions only work for a restricted set of program states, we invoke the theorem prover for subsequent verification to check if PS and $invs$ are valid for all states.

5.3 Leveraging Expression Trees

Despite the above two optimizations, the synthesis search space remains large. For example, in the context of S in Fig. 1a for which we need to synthesize two $invs$ and one PS , a depth-4 grammar, after removing the reduction operations, still presents around 100k potential solutions just for PS . TENSIR plays a significant role in the further pruning of this search space. The design of TENSIR operators effectively bridges the gap between high-level tensor operations and the loop-based paradigm commonly used for computing on tensors. This property of TENSIR allows us to leverage the expression-tree-based filtering approach, which we describe next, to efficiently prune the synthesis search space.

Our approach starts with the static analysis of S to identify the computations performed; the static analysis pass emits an expression tree that represents the computation. For example, the pre-order traversal of the expression tree for S from Fig. 1a is: (- (+ b a) (/ (* b a) 255)). In Fig. 6, this results in the pruning of `tensor_scalar` and `ite(cond5, m4, m4)` at the top-level (shown in teal) and similarly operators at other depths (m_4, m_3) are filtered. The generated expression tree is then transformed into an abstract expression tree, where variables and constants are replaced with placeholders, resulting in a synthesis template.

The abstract expression tree for S is then: $(- (+ \text{var} \text{ var}) (/ (* \text{var} \text{ var}) \text{lit}))$. This abstract expression tree guides TENSPLIER in identifying the sequence of TENSIR operators. In this example, TENSPLIER deduces the sequence of operators from the tree as: $t_t(t_t(\text{var}, \text{var}, +), t_s(t_t(\text{var}, \text{var}, *), \text{lit}, /), -)$, where var and lit are variables and literals to be synthesized, respectively.

Our expression trees are amenable to vectorized operations, which simultaneously perform the same computation on multiple data elements. Specifically, each level of the tree corresponds to an operation with the branches indicating data flow. In the example expression shown above, we see element-wise subtraction, addition, scalar division, and element-wise multiplication orchestrated such that it aligns with the vectorized execution of the original computation.

This approach is not confined to specific operators but is adaptable to a range of operations in TENSIR. It can identify constructs like if-else blocks, where *ite* arguments are determined using the same expression tree strategy, allowing the synthesis process to determine the optimal sequence of operators within the constructs. This flexibility extends to reduction operators and other complex operations, aiding in the synthesis of efficient operational sequences.

5.4 Constraining Variables

The final optimization is to pinpoint specific variables (*vars* such as a, b in Fig. 1a) and literals (*lits* such as 255 in Fig. 1a) to be used in the grammar. Specifically, we constrain the variables to the set of live variables and also constrain constants to the set of constants that have appeared in the program. This strategy simplifies the computational task, avoiding the complexity of synthesizing a complete depth-4 operator sequence. By leveraging our expression tree-based approach, the search space reduces to 64 expressions, and the synthesizer promptly yields the correct solution within 76 secs: $t_t(t_t(b, a, +), t_s(t_t(b, a, *), 255, /), -)$.

5.5 Overall Synthesis Algorithm

The algorithm described in Fig. 7 summarizes the synthesis phase in TENSPLIER. This phase is used to search for the bodies of PS and invariants which satisfy the VC. The synthesis is an iterative process conducted over multiple rounds, assuming a filtered search space leveraging type-based and expression tree optimizations described earlier. We start with the tensor bound size set to 2 which corresponds to restricting the program states optimization. In each round, we invoke Rosette’s search algorithm (line 5) to generate candidates for PS and $invs$. Upon obtaining a solution, the candidate undergoes validation against the VC for all program states, as the synthesis phase only checks within a constrained set of program states. We invoke a verifier (CVC5) (line 8) to perform this check. If the verifier yields “UNSAT,” the generated candidates are correct. Conversely, if “SAT” is returned, indicating incorrect candidates (line 11), the VC is augmented with blocking constraints. These constraints state that the generated PS or $invs$ in next round should differ from those in the previous rounds. This iterative process continues for a specified number of rounds (`max_rounds`) before incrementing the tensor bound sizes. In cases where Rosette’s search algorithm does not produce a solution initially (line 13), indicating an overly restrictive grammar, the initial grammar is expanded to include additional options for both PS and invariants, such as choices for loop bounds, indexing, and operator sequences. We keep a separate timer (not shown in Fig. 7) that maintains a maximum time bound for the entire synthesis process.

```

1  def synthesis_algorithm(spec, tensor_size_bound, holing_grammar, search_algorithm,
2      verifier, max_rounds, timeout):
3      r = 0 # rounds within one list bound
4      #bounded synthesis optimization
5      while r < max_rounds:
6          ps_inv = search_algorithm(spec, holing_grammar) #rosette
7          if ps_inv is not None:
8              ps_r, inv_r = ps_inv
9              if verifier(specification, ps_r, inv_r) == "UNSAT": return ps_r, inv_r
10             else:
11                 spec = spec and (ps != ps_r) and (inv != inv_r) #add blocking constraint
12                 r += 1
13             else:
14                 expand_holing_grammar(holing_grammar)
15             # Increment tensor size bound
16         return synthesis_algorithm(spec, tensor_size_bound + 1, holing_grammar,
17             search_algorithm, verifier, max_rounds, timeout)

```

Figure 7 TENSPILER synthesis algorithm.

6 Experiments

We evaluate TENSPILER’s effectiveness in converting code into various tensor processing infrastructures using 10 loop-based real-world benchmark sets: **blend**, **Llama** [24], **blas**, **darknet**, **dsp**, **dspstone**, **makespeare**, **mathfu**, **simpl_array**, and **utdsp**. The **blend** benchmarks focus on image processing kernels, the **Llama** benchmarks contain traditional deep learning applications, and the rest 8 are all sourced from various existing software libraries, such as the BLAS linear algebra library and the TI signal processing library, and are used recently to evaluate C to TACO translations [25]. This combination of benchmarks allows for a comprehensive assessment of TENSPILER’s effectiveness and adaptability across diverse domains and programming paradigms.

1. Used in prior work [6], the **blend** benchmarks consist of 12 functions dedicated to point-wise image blending operations – a fundamental aspect of image processing known for diverse visual effects. These functions span 180 lines of C++ code, and 10 are characterized by doubly-nested loops, which are common in image processing algorithms.
2. **Llama** benchmarks are derived from the C++ based inference code of Llama2 [24], an open-source LLM from Meta. We labeled the portion of code to be lifted from the source code without doing any extensive syntax or code logic edits. These benchmarks include 11 functions capturing essential operations like computing activations, attention mechanisms, and layer norms. They total around 106 lines of code, with 2 functions incorporating doubly-nested loops.
3. **blas**: 2 benchmarks from the BLAS [13] linear algebra library.
4. **darknet**: 10 neural network functions sourced from the Darknet [2] deep learning framework.
5. **dsp**: 12 signal processing functions from the TI library [4].
6. **dspstone**: Kernels from the DSPStone suites [38] that target digital signal architectures.
7. **makespeare**: Programs originally from Rosin [31] that manipulate integer arrays.
8. **mathfu**: 11 mathematical functions extracted from the Mathfu library [3].
9. **simpl_array**: 5 functions for computations on integer arrays originally from prior work [35].
10. **utdsp**: Kernels from the UTDSP suite [33] that targets digital signal architectures.

6.1 Evaluation Setup

The synthesis and verification phases for all benchmarks are executed on a MacBook Pro 2 GHz Quad-Core Intel Core i5 Processor with a timeout of 1 hour. After lifting the code to TENSIR, the code generator, as explained in Sec. 4.3, generates executable code for each target backend. In the next section, we first describe the datasets used for evaluating the performance of lifted benchmarks. Then, we describe each target backend used for executing these benchmarks.

6.1.1 Datasets for Evaluation

To mimic real-world settings, we evaluate the translated code on actual datasets instead of generating random inputs.

For the **blend** image processing benchmarks, **blas**, **darknet**, **dsp**, **dspstone**, **makespeare**, **mathfu**, **simpl_array**, and **utdsp**, we source images from ImageNet [17], a large-scale image dataset. We process these images as grayscale to ensure pixel values fall within the appropriate range. For benchmarks with 1-D tensor inputs, we flatten the images before feeding them as inputs and pass them in as they are for 2-D tensor inputs. For the blending layers in the blend benchmarks, we generate random pixel values from 0 to 255. We randomly select a set of 10,000 images from the dataset for evaluation.

For the **Llama** benchmarks, we evaluate the synthesized code using weights from Vicuna [16], an open-source LLM with similar model size as Llama2.⁶ Some kernels operate over model weights, for which we directly use the weight matrices from Vicuna. For kernels operating over inputs, we simulate embeddings by creating random 32-bit float vectors within the range [0, 1). The evaluation primarily uses the 33B-parameter Vicuna model; however, for evaluating the MLX framework, the 7B-parameter version is used due to memory limitations.

6.1.2 Target Software Frameworks and Hardware Accelerators

The core objective of TENSPILER is to translate sequential programs to a spectrum of diverse target DSLs, which can then be executed on conventional CPUs, GPUs, or specialized hardware accelerators. Although finding the optimal target DSL for the given input program would be an interesting feature in TENSPILER, currently TENSPILER is designed to provide users with the flexibility of choosing their preferred environment.

For our experimental evaluation, we choose 6 different target DSLs as we mentioned in Sec. 4.3: **NumPy**, **TensorFlow**, **PyTorch**, **MLX**, **TPC-C**, and **Gemmini**. We believe that our comprehensive selection of DSLs is necessary to test the robustness of TENSPILER.

The TENSIR design greatly simplified this process as NumPy, TensorFlow, PyTorch, and MLX have similar APIs, and each of these DSLs uses only 200 lines of code for generating executable code.

To establish a baseline for execution performance, we compile C++ code for all the benchmarks using `gcc-8.3.0` with `-O3` flag and then run them on an Intel Xeon 8380 CPU. Given that each DSL is tailored to enhance performance on specific hardware, we conduct evaluation across five distinct platforms: the Intel Xeon 8380 CPU, Nvidia V100 GPU, Apple M1 Pro, Intel Gaudi 2 processor, and the Gemmini accelerator.⁷ In all, we utilized 7 different

⁶ We did not use Llama2 model weights as they are not publicly available.

⁷ Due to lack of physical device, Gemmini evaluations are done on a simulator with limited computing power and no file system support. Thus, it is compared with smaller random inputs.

DSL-hardware device combinations for our experiments: (1) NumPy-CPU, (2) TensorFlow-V100, (3) PyTorch-V100, (4) MLX-Apple M1 (5) TPC-C-Gaudi, (6) PyTorch-Gaudi, and (7) Gemmini. This comprehensive mapping of each backend to its corresponding device enables us to accurately assess the benefits TENSPILER provided through lifting.⁸

6.2 Synthesis Timings

In this section, we present the time TENSPILER takes to synthesize and verify solutions for each of our benchmarks. During the synthesis phase, we apply all the optimizations mentioned in Sec. 5 with a 1 hour timeout. TENSPILER synthesizes the correct translations for all the benchmarks under **15 mins**.

Fig. 8 illustrates the synthesis performance for our 10 benchmark suites. Fig. 8a illustrates the synthesis performance for the **blend** benchmarks. All but three benchmarks are synthesized in one synthesis (and verification) iteration, with an average synthesis time of 40.7 seconds and a median synthesis time of 2.357 seconds. Single-loop benchmarks synthesize with an average of 2.17 seconds and a median of 1.92 seconds. Double-loop benchmarks, on the other hand, have an average of 128.91 seconds and a median of 22.74 seconds for synthesis.

The three benchmarks that take more than one round of synthesis are **softmax1**, **transformer1**, and **transformer2** from the **LLama** suite. **softmax1** fails to synthesize within one round because the initial grammar is overly restrictive for its loop invariant. **transformer1** and **transformer2** involve complex indexing constraints for their invariants, initially leading to spurious solutions with tensor size limit of 2. **transformer2** finds the correct solution after 6 tries, while **transformer1** exhausts the maximum number of tries (10) with tensor size 2. We then increase the tensor size limit to 3 for **transformer1** and a correct solution is generated within 3 rounds.

6.2.1 Analysis

We observe that synthesis difficulty is correlated to both the number of loops and the complexity of the TENSIR solution. For example, the **dot** benchmark in the blas benchmark set has a single loop. Its TENSIR solution is `reduce_sum(t_t(a, b, *))`, which has two operators and only two arguments, **a** and **b**, to be synthesized. This benchmark synthesizes in around two seconds. On the other hand, the **transformer_part1** benchmark from the **LLama** benchmark set has a doubly-nested loop and a complex solution with six operators. All arguments to these operators must be synthesized, with some requiring 3 expressions such as `head * head_size + head_size` and `sqrt(head_size * 1)`. This benchmark takes around 1300 seconds to synthesize.

In addition to synthesis challenges, we recognize that the tree approach may restrict the ability of TENSPILER to synthesize different solutions. However, through manual verification, we confirm that TENSPILER consistently generates optimal solutions across our benchmarks. We evaluate the solutions using expression length as the cost function. Generally, shorter expressions mean fewer function calls and thus lower execution cost. To illustrate, we use the **linear_dodge** example from the **blend** benchmarks for which the synthesized solution is as follows:

```
def linear_dodge(a, b): t_t(a, b, +)
```

⁸ The exact configurations of all the hardware devices and their software environments are described in Appendix C in the extended version of this paper[29]

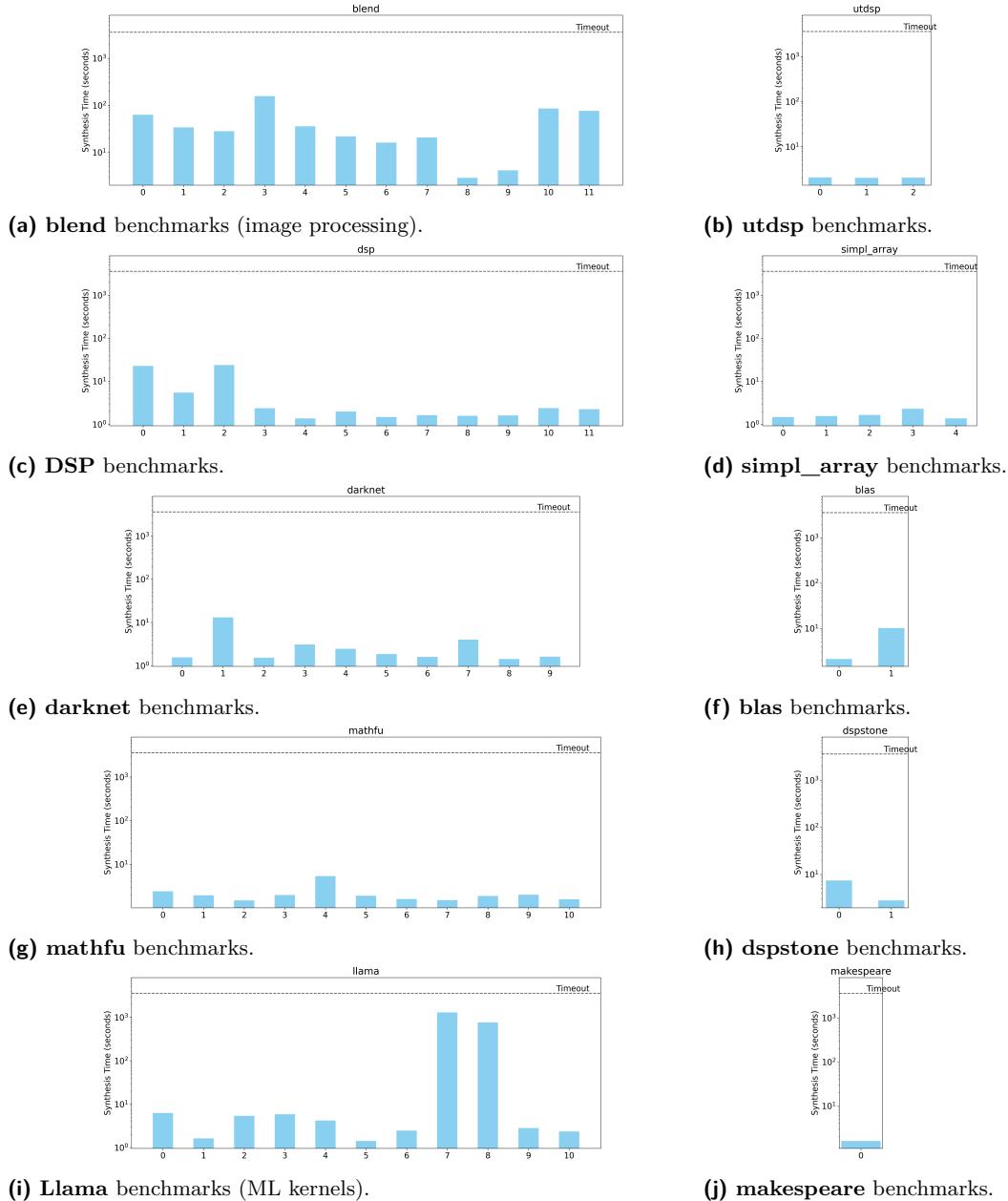


Figure 8 Synthesis timings for all the benchmark suites. Benchmark name legend in Appendix B in the extended version of this paper[29].

After relaxing the constraint on the tree structure, we were able to synthesize a different solution as shown below:

```
def linear_dodge(a, b): t_t(a, t_s(b, -1, *), -)
```

The latter solution is longer in length, and involves **2** tensor operations – a **tensor_tensor** operation and a **tensor_scalar** operation – as opposed to **1** **tensor_tensor** operation synthesized using the tree approach. Therefore, it is less optimized. In addition, the tree approach also speeds up the synthesis process as shorter expressions are easier to synthesize.

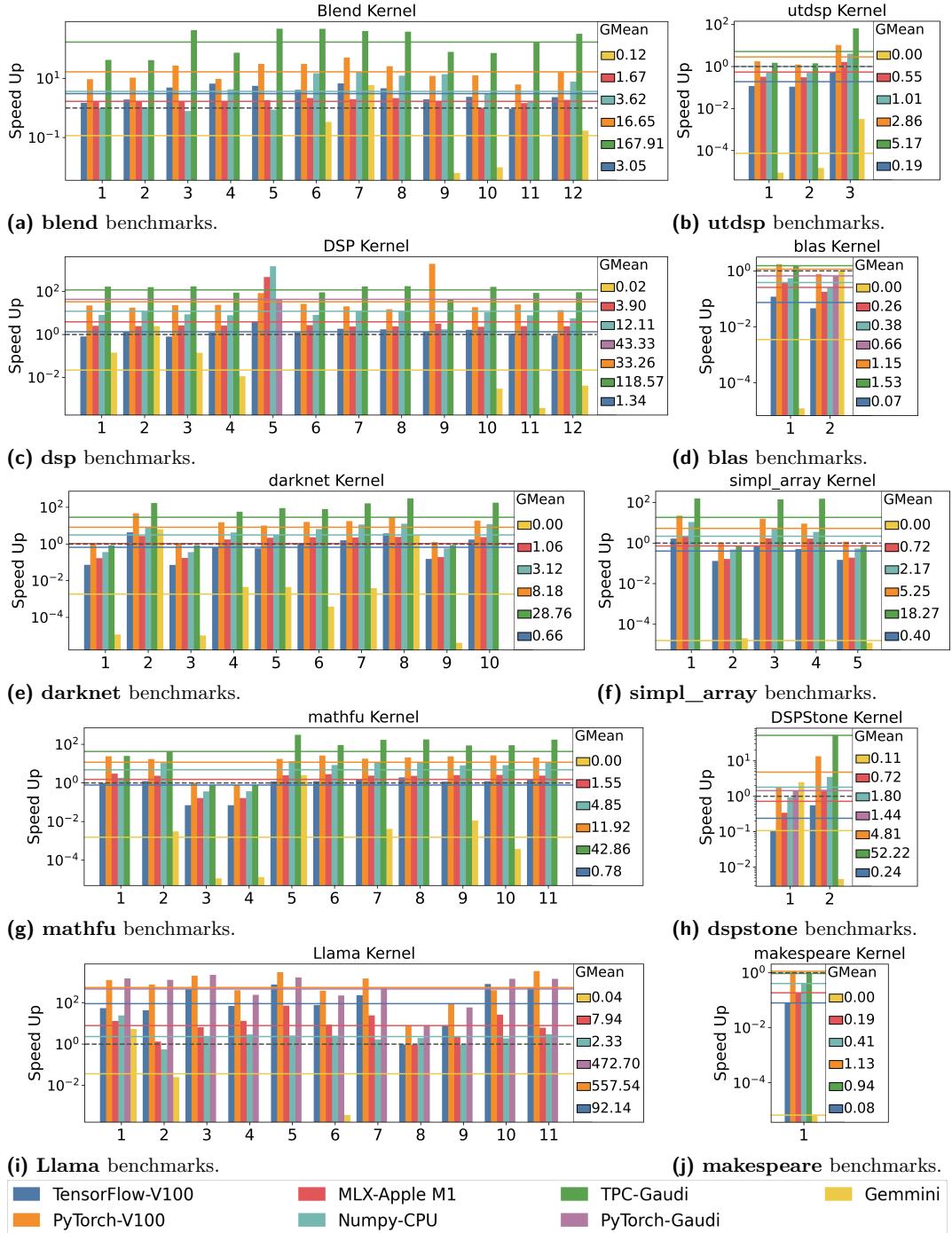


Figure 9 Kernel speedup over baseline. Benchmark name legend in Appendix B in the extended version of this paper[29].

6.3 Performance Timings

In this section, we evaluate the performance of the original input code – sequential C++ programs compiled with `gcc -O3` – by comparing them with their translated versions executed across different target backends as detailed in Sections 6.1.2 and 6.1.1.

Kernel Performance. Kernel performance focuses on computation time excluding data transfer overhead. We see significant improvements as illustrated in Figures 9, with an average speedup of **105.1 \times** across all benchmarks. Notably, the Gaudi 2 processor demonstrates an exceptional speedup of **241 \times** , highlighting the advantages of migrating legacy code to newer hardware platforms. For other backends such TensorFlow, PyTorch, MLX and NumPy we see speedups of **46 \times** , **244 \times** , **10.5 \times** and **26.12 \times** respectively.

However, compatibility issues can emerge with certain backends. For example, the Gemmini accelerator does not support certain operations in our TENSIR like `tensor_tensor` element-wise multiplication, `slice`, and `tail`. To address this, we only translate supported TENSIR operations from the synthesized *PS*, and default to running the unsupported operations using sequential C on CPUs. Out of 69 benchmarks, 41 are translatable to Gemmini’s instruction set architecture (ISA), yet only 10 can be fully expressed using Gemmini instructions alone. Challenges are notable in benchmarks like `screen_blend` (Fig. 9a), where element-wise vector multiplication must fallback to execution on a less powerful CPU. Furthermore, most Gemmini’s instructions require square matrices inputs. This means that we need to pad vector inputs to square matrices before being able to utilize Gemmini’s instructions, effectively squaring the data volume to be processed. This results in varied performance as shown in Fig. 9i and Fig. 9e.

End-to-end Performance. While frameworks and accelerators deliver substantial kernel performance enhancements, a comprehensive assessment must account for end-to-end benchmark times, encompassing initial setup and data movement between the host (CPU) and the accelerator device. Our focus here is on data transfer (TensorFlow, PyTorch, and Gaudi processor) and memory management (C++). As illustrated in Fig. 10, we again observe an overall speedup, averaging **9.7 \times** . In particular, CPU libraries like NumPy and MLX show more improvements with the notable advantage of avoiding transferring data to specialized hardware. These benchmarks, involving the processing of 1D or 2D character vectors, benefit largely from C++’s efficient handling of contiguous data structures. Meanwhile, Gaudi 2 drivers encounter performance bottlenecks due to the overheads associated with hardware initialization and frequent small data transfers. This significant upfront cost, especially pronounced in small-scale data operations, leads to a much less announced speedup. We believe such a phenomenon is uncommon in real-world use cases such as training deep learning models, due to techniques like batch processing or pipelining to minimize data transfers or to overlap computations with communications, thereby reducing or hiding transfer overhead and enhancing overall efficiency.

Compare Against Pattern Matching-Based Compilers. As outlined in Sec. 1, TENSPILER is designed to address the limitations inherent in traditional compilers that rely on pattern matching to compile. Such compilers are resource-intensive to develop and prone to errors. To the best of our knowledge, no existing compiler matches the breadth of DSL support offered by TENSPILER. However, specialized compilers, such as Numba [28], have been introduced for accelerators like GPUs. Numba leverages LLVM IR to generate GPU-accelerated code from Python code, making it a suitable candidate for comparison.

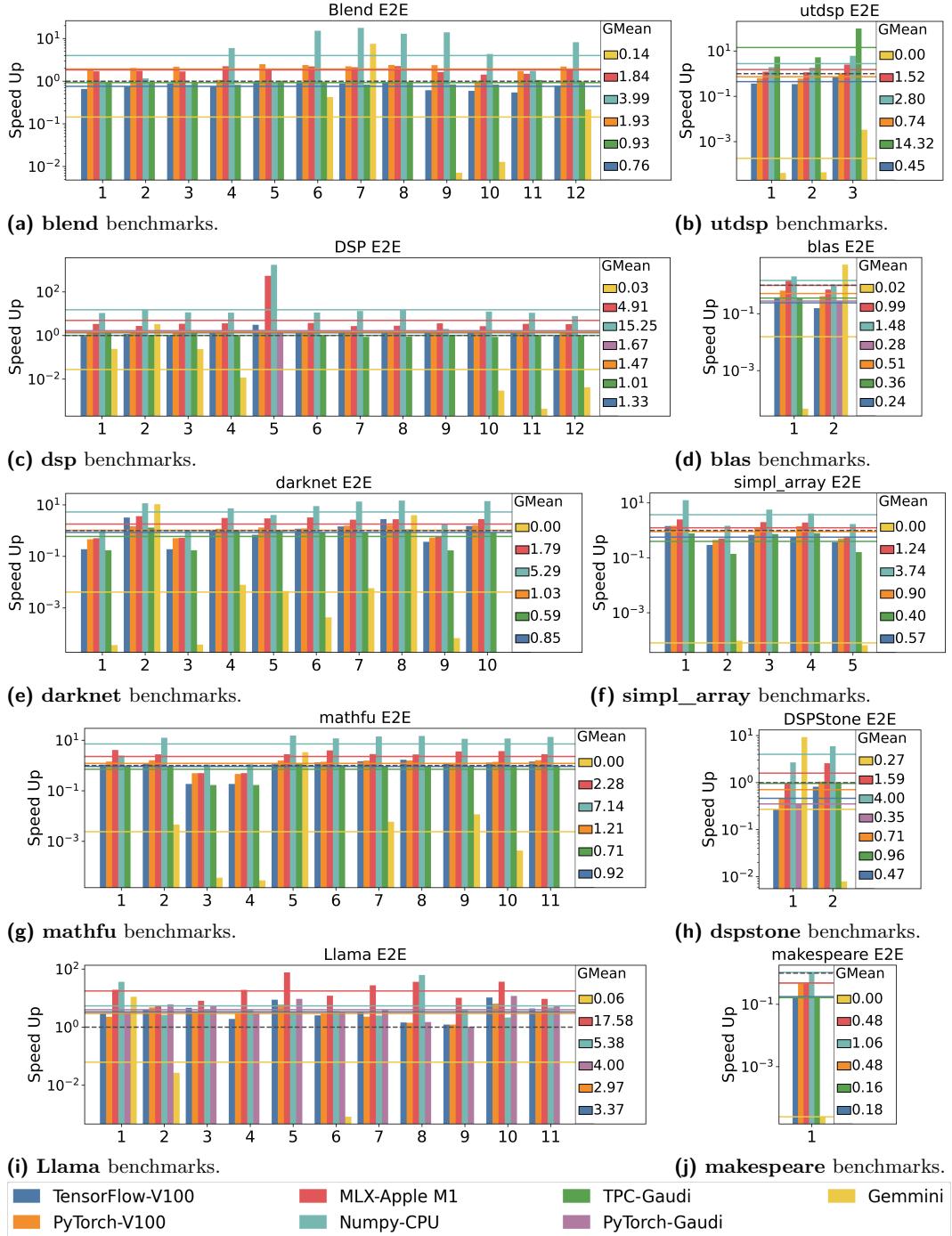


Figure 10 E2E speedup over baseline. Benchmark name legend in Appendix B in the extended version of this paper[29].

```

1  vector<float> matmul(vector<vector<float>> weight, vector<float> input) {
2      vector<float> output;
3      int m = weight.size();
4      int n = input.size();
5      for (int row = 0; row < m; row++) {
6          float curr = 0;
7          for (int col = 0; col < n; col++) {
8              curr += weight[row][col] * input[col];
9          }
10         output.push_back(curr);
11     }
12     return output;
13 }
```

(a) Original `matmul` function in C++.

```

1  @cuda.jit()
2  def matmul (weight, input, res):
3      m = len(weight)
4      n = len(input)
5      for i in range(m):
6          curr = 0
7          for j in range(n):
8              curr += weight[i][j] * input[j]
9          res[i] = curr
10 
```

(b) Numba kernel annotated version of `matmul`.**Figure 11** Manually rewritten Numba example.

For benchmarking purposes, we utilize the same datasets, test cases, and setup described previously in Sec. 6. Benchmarks are rewritten in Python and adapted to conform to CUDA kernel requirements by removing return statements, as shown in Fig. 11. Additionally, relevant data are cast to NumPy arrays as Numba focuses on optimizing code written against NumPy’s API. These syntactic requirements represent a limitation of Numba’s approach. In contrast, TENSPLIER operates directly on the original benchmark implementations.

Experimental results demonstrate that GPU-based PyTorch and TensorFlow code generated by TENSPLIER performs, on average, 1.87 \times faster than code annotated with Numba. Remarkably, while Numba benefits from years of development by expert engineers, TENSPLIER achieves superior performance with only 200 additional lines of code dedicated to code generation. A closer examination of the compiled PTX assembly code for the `matmul` benchmark, which shows a 2.6 \times speedup, reveals that the Numba-generated code lacks the use of advanced instructions and techniques such as fused multiply-add (FMA), tiled-based computation models, or shared memory,⁹ which are crucial for peak performance. These techniques are standards in PyTorch and TensorFlow with optimized kernels. In contrast, Numba requires extensive manual tuning to implement, evident in the more complex and faster `matmul` example in its documentation.¹⁰ TENSPLIER, by automatically recognizing and translating matrix multiplication operations to leverage the pre-optimized kernels, avoids the complexities of manual code optimization while achieving high performance.

6.4 Ablation Study

In our ablation study, we evaluate using our benchmark suites the effectiveness of the optimizations (described in Sec. 5) in making synthesis scale.

⁹ See Appendix D in the extended version of this paper[29] for the PTX code.

¹⁰ For the detailed example of an optimized `matmul` function with shared memory for Numba, see <https://numba.readthedocs.io/en/stable/cuda/examples.html#id30>

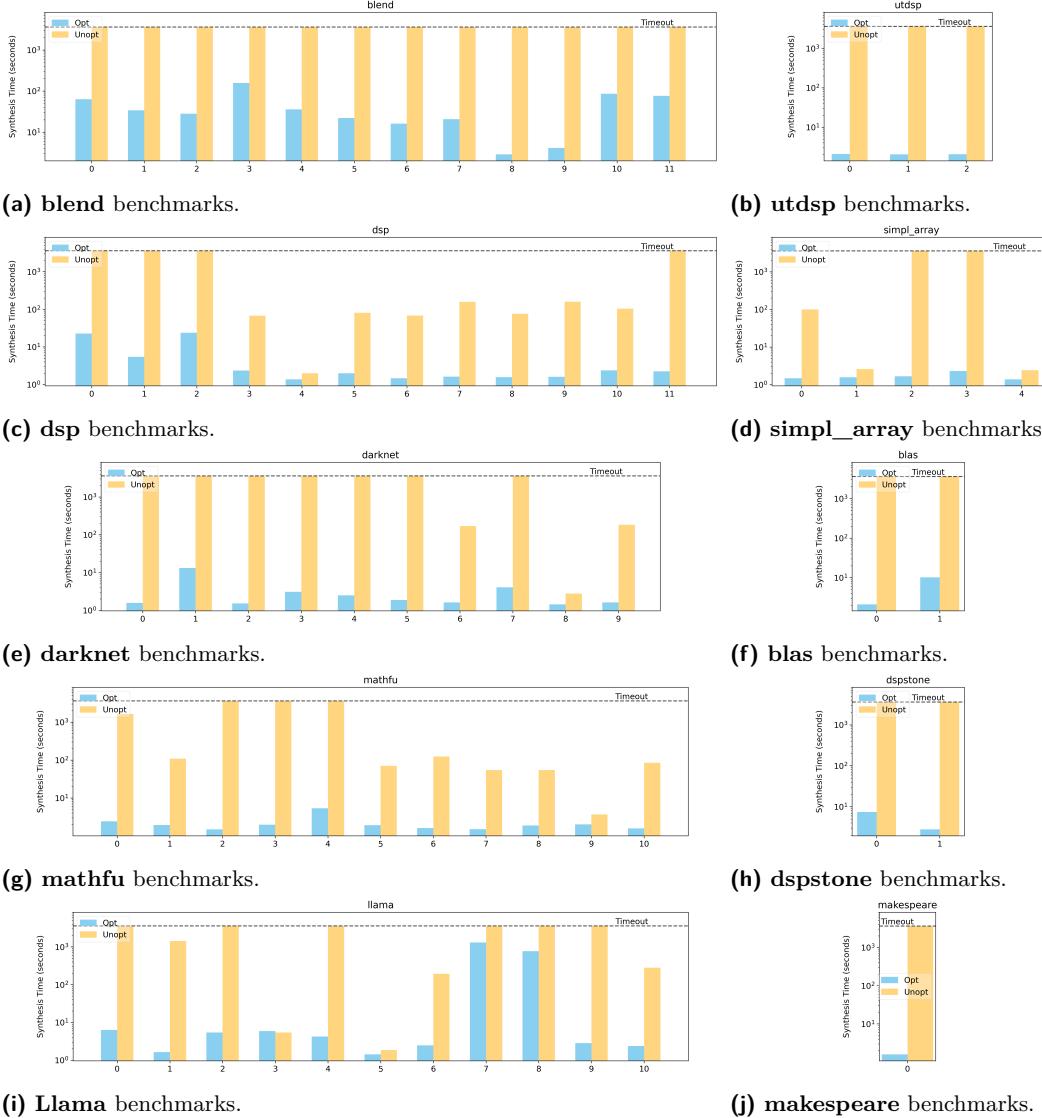


Figure 12 Synthesis timings for all the benchmarks with and without TENSPILER’s tree-based optimization. Benchmark name legend in Appendix B in the extended version of this paper[29].

Bounded Synthesis. In this experiment, we keep the type-based filtering and tree approach while removing the incremental bounded synthesis optimizations. We start with a static tensor bound of 4 instead of the incremental approach. With this, **6** of the **12** **blend** benchmarks time out. In addition, benchmarks involving 2D tensors that do not time out see an average of **36.75 \times** slowdown.

Tree Approach. For this experiment, we include type-based filtering and remove the expression tree approach for grammar filtering. We assume a fixed depth for the grammar, i.e., including all operators up to the specified depth, and increase it upon synthesis failure (starting at depth 1). Without static analysis, no assumptions are made about the operators, slice indices, variables, or constants, necessitating their synthesis. Unlike the tree approach with a fixed number of placeholders, this approach exhibits scalability issues as the number of grammar choices increases exponentially with depth. Therefore, only benchmarks with depth 1 and 2 expressions could be successfully synthesized.

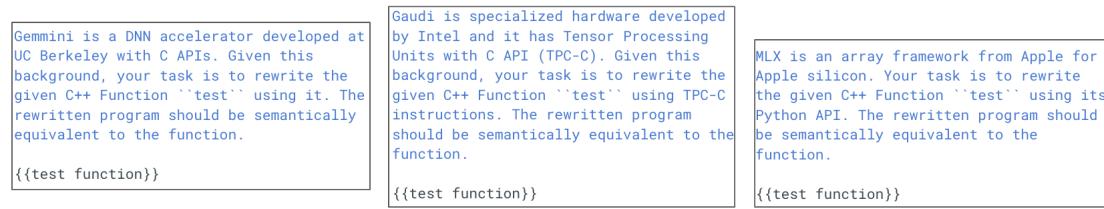


Figure 13 Prompts for LLM.

As illustrated in Fig. 12, without the tree based optimization, **42** out of the total **69** benchmarks timed out. In particular, all benchmarks of the **blend**, **blas**, **dspstone**, **makespeare**, and **utdsp** suites timed out. For the benchmarks that succeed, the synthesis phase slows down by an average **101.55 \times** compared to the tree-based grammar filtering approach due to the need to synthesize additional expressions in both *PS* and *invs*.

6.5 Comparison with LLMs

LLMs have shown promising results in various programming-related tasks, such as code generation, translation, and testing. However, these models suffer from a lack of formal verification of the translated code and face challenges in adapting to new DSLs or backends that are not well-represented in their training corpus.

To test the capabilities of LLMs in generating code for new or low-resource DSLs, we prompt a state-of-the-art proprietary LLM Claude Opus [1] (our evaluation using other LLMs such as GPT4 shows similar results). We selected three backends for this experiment: MLX, a completely new DSL, Gaudi, and Gemmini, which are not well represented in the training corpus of these models. We prompted Claude Opus to generate code for these DSLs for the `linear_dodge` benchmark from the **blend** suite. The prompt instructions are shown in Fig. 13. In Fig. 14 we show the code generated by the LLM for the three prompts.

Upon analysis, we found that all three generated programs were incorrect. The Gemmini-generated code in Fig. 14a partially uses the correct APIs (`mvin`, `mvouts`), but the computation with `config_ex` is incorrect. For the Gaudi-generated code in Fig. 14b, the model hallucinates the TPC-C library, which does not exist in the actual Gaudi programming model. The MLX-generated code in Fig. 14c has the correct call to the library function `add`, but the imports are incorrect, making the code non-functional. In addition to the generated code being incorrect, it is challenging to verify these outputs formally as syntactically translating the generated code to SMT-LIB is not trivial. The experiment highlights two significant challenges in generating verified code using LLMs mentioned earlier. In contrast, TENSPILER, which uses a verified lifting-based approach, can easily handle these challenges. LLMs cannot be directly prompted to generate code in new DSLs. LLMs could potentially be fine-tuned or prompted with few-shot learning to generate code in an IR, which can then be utilized within the TENSPILER's framework for verification; however, we leave this as future work.

6.6 Extension to Higher-Dimensional Tensors

Our benchmarks only involve 1D and 2D tensors, as most operations are performed on images (the **blend** benchmarks) and weight matrices (the **LLama** benchmarks). In this section, we demonstrate that TENSPILER can be extended to support higher-dimensional tensors with the generalizability of TENSIR and the synthesis optimizations discussed in Sec. 5. Specifically, we extend TENSIR to accommodate 3D tensors and all corresponding element-wise operations.

```
#include <gemmini.h>

void linearDodge8_gemmini(uint8_t* base, uint8_t* active, uint8_t* out, int height, int width) {
    gemmini_config_ex(BANK_SIZE, BANK_NUM, ACCEL_NUM);
    gemmini_flush(0);
    elem_t A[height * width];
    elem_t B[height * width];
    elem_t C[height * width];
    for (int i = 0; i < height * width; i++) {
        A[i] = base[i];
        B[i] = active[i];
    }
    gemmini_config_st(A, height * width);
    gemmini_config_st(B, height * width);
    gemmini_config_ld(C, height * width);

    gemmini_mvin(A, 0);
    gemmini_mvin(B, 1);
    gemmini_config_ex(ELEM_ADD, 0, 0, 0);
    gemmini_mvout(C, 0);

    for (int i = 0; i < height * width; i++) {
        out[i] = (uint8_t)C[i];
    }
    gemmini_fence();
}

#include <tpc.h>

void linearDodge8_tpc(Buffer<uint8_t,2> base, Buffer<uint8_t,2>
active, Buffer<uint8_t,2> out) {
    int height = out.height();
    int width = out.width();
    tpc::nd_range<2> nd_range({height, width}, {1, 1});
    tpc::parallel_for(nd_range, [&](tpc::nd_item<2> item) {
        int row = item.get_global_id(0);
        int col = item.get_global_id(1);
        tpc::vector<uint8_t, 16> base_vec = tpc::load<uint8_t,
16>(base.data() + (row * width + col));
        tpc::vector<uint8_t, 16> active_vec = tpc::load<uint8_t,
16>(active.data() + (row * width + col));
        tpc::vector<uint8_t, 16> result_vec = base_vec + active_vec;
        tpc::store<uint8_t, 16>(out.data() + (row * width + col),
result_vec);
    });
}
```

(a) LLM generated Gemmini Code.

```
import mlx

def linear_dodge_8(base, active):
    out = mlx.add(base, active)
    return out.astype(mlx.uint8)
```

(c) LLM generated MLX Code.

(b) LLM generated Gaudi Code.

Figure 14 LLM generated code for the prompt.

Additionally, we adapt the operator restriction optimization (introduced in Sec. 5.1) to apply to 3D tensors. When the source program returns a 3D tensor, our grammar is restricted to include only element-wise 3D tensor operations. We also retain the program state restriction optimization technique from Sec. 5.2. Furthermore, we extend our support to leverage expression trees performed on individual elements in tensors, as detailed in Sec. 5.3, to guide the search for vectorized operations within 3D tensor spaces.

We evaluate TENSPILER’s synthesis optimizations on artificial benchmarks involving 3D tensors. We create these benchmarks by combining random element-wise operations. The maximum depth of these benchmarks is chosen to be 5 to match that of all our existing real-world benchmarks, as described in Sec. 6. Results in Fig. 15 show that the synthesis time grows linearly with the depths of the benchmarks. The depth-1 benchmark synthesizes the fastest in 6 seconds, while the depth-5 benchmark takes the longest, in 184 seconds.

The sharp increase in timing for depth 5 expressions in Fig. 15 is due to the number of expressions we are synthesizing and their complexity. A benchmark with 3 loops involves synthesizing 3 invariants and 1 post-condition, each with expression sizes up to depth 5. Despite these challenges, we easily extend TENSPILER’s optimizations and synthesize these benchmarks well within the 1 hour timeout. As future work, to further scale TENSPILER’s synthesis algorithm for handling more complex benchmarks, we could explore strategies such as guiding the search process using machine learning techniques, implementing bottom-up synthesis starting with inner loops first, performing bounded synthesis with unrolled loops, and combining these approaches with TENSPILER’s current synthesis optimizations.

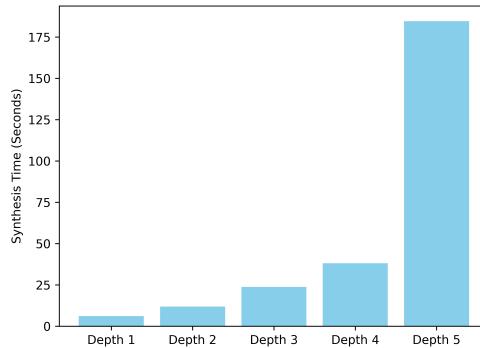


Figure 15 Synthesis timings for artificial 3D tensor benchmarks.

7 Related Work

Verified Lifting. Verified lifting uses program synthesis to translate code instead of designing traditional pattern-matching compilers, and has been used across application domains [15, 5, 6, 19, 21]. Adapting these prior compilers for translation to tensor operations is nontrivial. TENSPILER introduces a novel tensor algebra-based to make synthesis efficient, and supports a diverse set of backends.

Code Translators. TENSPILER differs from other code translation approaches. While symbolic methods like pattern-matching compilers [30] face challenges with the error-prone nature of their rules, TENSPILER uses a search-based approach to avoid these complexities. Neural techniques [32, 26], treat translation as a machine translation task but struggle to ensure correctness. In contrast, TENSPILER uses a theorem prover to guarantee semantic equivalence between the translated and source code. More recently, despite the success of LLMs in programming tasks, they are unable to translate code to unfamiliar frameworks or custom hardware ISAs. TENSPILER’s approach of searching in an TENSIR and using simple rules for translation makes it easy to support new backends.

Intermediate Representations. LLVM [22], MLIR [23] and TACO’s IR [20] are examples of IRs that generate code to multiple backends. LLVM in addition can generate optimized code for various hardware targets. MLIR introduces “dialects,” allowing specific optimizations for different domains or hardware targets. Despite their versatility, LLVM and MLIR were originally designed for traditional pattern-matching compilers, posing challenges for search-based compilers due to their extensive set of operators. In contrast, TENSIR is designed for expressing tensor operations to be used in search-based compilers. As discussed, TENSIR enables efficient lifting, verification, and code generation.

8 Conclusions

We presented our experience in building TENSPILER, a compiler that leverages verified lifting to transpile code to leverage tensor processing infrastructures. At the core of TENSPILER is TENSIR which concisely captures various tensor computations. TENSPILER efficiently translates all 69 real-world benchmarks and can generate code to be executed on 6 different software and hardware backends. The generated code achieves an average speedup of **105 \times** for kernel and **9.65 \times** for end-to-end execution compared to the input.

References

- 1 Claude Model. <https://www.anthroscopic.com/news/claudie-3-family>. [Online].
- 2 Darknet. <http://pjreddie.com/darknet/>. [Online].
- 3 Mathfu. <https://github.com/google/mathfu>. [Online].
- 4 Texas Instrument Digital Signal Processing (DSP) Library for MSP430 Microcontrollers. <https://www.ti.com/tool/MSP-DSPLIB>. [Online].
- 5 Maaz Bin Safeer Ahmad and Alvin Cheung. Automatically leveraging mapreduce frameworks for data-intensive applications. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1205–1220. ACM, 2018.
- 6 Maaz Bin Safeer Ahmad, Jonathan Ragan-Kelley, Alvin Cheung, and Shoaib Kamil. Automatically translating image processing libraries to halide. *ACM Trans. Graph.*, 38(6), November 2019. doi:10.1145/3355089.3356549.
- 7 Maaz Bin Safeer Ahmad, Alexander J. Root, Andrew Adams, Shoaib Kamil, and Alvin Cheung. Vector instruction selection for digital signal processors using program synthesis. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’22, pages 1004–1016, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3503222.3507714.
- 8 Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pages 1–8, 2013. doi:10.1109/FMCAD.2013.6679385.
- 9 SMT-LIB Authors. SMT-LIB Standard. <https://smtlib.cs.uiowa.edu/>. [Online].
- 10 Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength smt solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442, Cham, 2022. Springer International Publishing.
- 11 Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE ’05, pages 82–87, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/1108792.1108813.
- 12 Sahil Bhatia, Sumer Kohli, Sanjit A. Seshia, and Alvin Cheung. Building Code Transpilers for Domain-Specific Languages Using Program Synthesis. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, volume 263 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:30, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2023.38.
- 13 L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.
- 14 Alexander Brauckmann, Elizabeth Polgreen, Tobias Grosser, and Michael FP O’Boyle. mlir-synth: Automatic, retargetable program raising in multi-level ir using program synthesis. In *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 39–50. IEEE, 2023.
- 15 Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’13, pages 3–14, New York, NY, USA, 2013. ACM. doi:10.1145/2491956.2462180.

- 16 Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality, March 2023. URL: <https://lmsys.org/blog/2023-03-30-vicuna/>.
- 17 Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009. doi:[10.1109/CVPR.2009.5206848](https://doi.org/10.1109/CVPR.2009.5206848).
- 18 C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
- 19 Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. Verified lifting of stencil computations. *SIGPLAN Not.*, 51(6):711–726, June 2016. doi:[10.1145/2980983.2908117](https://doi.org/10.1145/2980983.2908117).
- 20 Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017. doi:[10.1145/3133901](https://doi.org/10.1145/3133901).
- 21 Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, and Joseph M. Hellerstein. Katara: synthesizing crdts with verified lifting. *Proc. ACM Program. Lang.*, 6(OOPSLA2):1349–1377, 2022.
- 22 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, page 75, USA, 2004. IEEE Computer Society.
- 23 Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A compiler infrastructure for the end of moore’s law, 2020. arXiv:[2002.11054](https://arxiv.org/abs/2002.11054).
- 24 llamacpp. <https://github.com/leloykun/llama2.cpp/>, 2024. Accessed: 2024-01-19.
- 25 José Wesley de Souza Magalhães, Jackson Woodruff, Elizabeth Polgreen, and Michael F. P. O’Boyle. C2taco: Lifting tensor code to taco. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2023, pages 42–56, New York, NY, USA, 2023. Association for Computing Machinery. doi:[10.1145/3624007.3624053](https://doi.org/10.1145/3624007.3624053).
- 26 Benjamin Mariano, Yanju Chen, Yu Feng, Greg Durrett, and Işil Dillig. Automated transpilation of imperative to functional code using neural-guided program synthesis. *Proc. ACM Program. Lang.*, 6(OOPSLA1), April 2022. doi:[10.1145/3527315](https://doi.org/10.1145/3527315).
- 27 Apple mlx. <https://ml-explore.github.io/mlx/>, 2024.
- 28 Numba. <https://numba.readthedocs.io/en/stable/cuda/overview.html>, 2024.
- 29 Jie Qiu, Colin Cai, Sahil Bhatia, Niranjan Hasabnis, Sanjit A. Seshia, and Alvin Cheung. Tenspiler: A verified lifting-based compiler for tensor operations, 2024. arXiv:[2404.18249](https://arxiv.org/abs/2404.18249).
- 30 Cosmin Radoi, Stephen J. Fink, Rodric Rabbah, and Manu Sridharan. Translating imperative code to mapreduce. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA ’14, pages 909–927, New York, NY, USA, 2014. ACM.
- 31 Christopher D Rosin. Stepping stones to inductive synthesis of low-level looping programs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33(01), pages 2362–2370, 2019.
- 32 Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In Hugo Larochelle, Marc’Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL: <https://proceedings.neurips.cc/paper/2020/hash/ed23fbf18c2cd35f8c7f8de44f85c08d-Abstract.html>.

- 33 Mazen AR Saghir. *Application-specific instruction-set architectures for embedded DSP applications*. Citeseer, 1998.
- 34 Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In Marinho P. Barcellos, Jon Crowcroft, Amin Vahdat, and Sachin Katti, editors, *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, pages 15–28. ACM, 2016.
- 35 Sunbeam So and Hakjoo Oh. Synthesizing imperative programs from examples guided by static analysis. In *International Static Analysis Symposium*, pages 364–381. Springer, 2017.
- 36 Emina Torlak and Rastislav Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, Onward! 2013*, pages 135–152, New York, NY, USA, 2013. ACM. doi:10.1145/2509578.2509586.
- 37 Tensorflow xla. <https://www.tensorflow.org/xla/architecture>, 2024.
- 38 Vojin Zivojinovic. Dspstone: A dsp-oriented benchmarking methodology. *Proc. Signal Processing Applications & Technology, Dallas, TX*, 1994, pages 715–720, 1994.

Compiling with Arrays

David Richter  

Technische Universität Darmstadt, Germany

Timon Böhler  

Technische Universität Darmstadt, Germany

Pascal Weisenburger  

University of St. Gallen, Switzerland

Mira Mezini  

Technische Universität Darmstadt, Germany

The Hessian Center for Artificial Intelligence (hessian.AI), Darmstadt, Germany

Abstract

Linear algebra computations are foundational for neural networks and machine learning, often handled through arrays. While many functional programming languages feature lists and recursion, arrays in linear algebra demand constant-time access and bulk operations. To bridge this gap, some languages represent arrays as (eager) functions instead of lists. In this paper, we connect this idea to a formal logical foundation by interpreting functions as the usual negative types from polarized type theory, and arrays as the corresponding dual positive version of the function type. Positive types are defined to have a single elimination form whose computational interpretation is pattern matching. Just like (positive) product types bind two variables during pattern matching, (positive) array types bind variables with *multiplicity* during pattern matching. We follow a similar approach for Booleans by introducing conditionally-defined variables.

The positive formulation for the array type enables us to combine typed partial evaluation and common subexpression elimination into an elegant algorithm whose result enjoys a property we call maximal fission, which we argue can be beneficial for further optimizations. For this purpose, we present the novel intermediate representation *indexed administrative normal form* (A_iNF), which relies on the formal logical foundation of the positive formulation for the array type to facilitate maximal loop fission and subsequent optimizations. A_iNF is normal with regard to commuting conversion for both let-bindings and for-loops, leading to flat and maximally fissioned terms. We mechanize the translation and normalization from a simple surface language to A_iNF , establishing that the process terminates, preserves types, and produces maximally fissioned terms.

2012 ACM Subject Classification Software and its engineering → Domain specific languages

Keywords and phrases array languages, functional programming, domain-specific languages, normalization by evaluation, common subexpression elimination, polarity, positive function type, intrinsic types

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.33

Related Version Full Version: <https://arxiv.org/abs/2405.18242>

Supplementary Material Software (ECOOP 2024 Artifact Evaluation approved artifact):

<https://doi.org/10.4230/DARTS.10.2.18>

Software: <https://github.com/stg-tud/ainf-compiling-with-arrays> [32]

archived at `swh:1:dir:8e0e755d11e4e3e91fb05bf8df1a5c8bec0f553a`

Funding Timon Böhler: LOEWE/4a//519/05/00.002(0013)/95.

Pascal Weisenburger: Swiss National Science Foundation (SNSF, No. 200429).

Mira Mezini: LOEWE/4a//519/05/00.002(0013)/95; HMWK cluster project *The Third Wave of Artificial Intelligence* (3AI).

 © David Richter, Timon Böhler, Pascal Weisenburger, and Mira Mezini;
licensed under Creative Commons License CC-BY 4.0
38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 33; pp. 33:1–33:24



1 Introduction

Linear algebra computations are of rising importance due to their foundational role in neural networks and other machine learning systems. The fundamental unit of computation in linear algebra is the multidimensional array (or just *array* from now on). Linear algebra programs are full of computations that construct arrays from other arrays, such as element-wise sum, matrix multiplication, convolution, (transformer) attention, and more.

In functional programming we usually work with lists not arrays. Lists are inductively defined data types, and processed using recursion. Even element access on lists is implemented by recursion, so it has to traverse the list until the element is found, giving it running time¹ linear in the size of the list. Arrays, on the other hand, should have constant-time access and be processed using bulk operations. As such arrays do not fit into the usual pattern of inductive data types. A number of functional programming languages that aim to facilitate programming of array computations have been proposed [12, 37, 31]. Aiming for high expressivity with few language constructs, they leverage the idea that arrays can be represented as functions [12, 37, 31], or that arrays are eager functions [28].

Functions are lazy in the sense that a function definition does not perform any computation and the function body is only executed when a function is applied. Arrays are eager in the sense that all contents of an array are evaluated during construction, and array access does not perform further computation. Prior work had this intuition on the duality between arrays and functions, and here we ground that correspondence on proof-theoretic concepts, which explain why arrays are eager and functions are lazy and yield further convenient consequences.

A key insight of our work is that arrays can be interpreted as a positively polarized version of the function type. The connection between the positive and negative formulation of data types and the computational interpretation of elimination forms as pattern matching has been developed in the context of polarized type theory and focused logic [39, 1, 9, 10, 8, 19, 40]. Thus, similarly to how pattern matching on a binary tuple introduces two variables, the computational content of pattern matching on an array of size N is the introduction of N -many variables. Further, we explore pattern matching on Booleans by introducing *conditional variables*. This insight provides the foundation for the design of the *indexed administrative normal form* ($A_i\text{NF}$), an intermediate representation for array computations. We also present a simple surface array language, called POLARA, and show how these changes together (the positive formulation for the array type and a negative presentation of Booleans) enables us to combine *typed partial evaluation* (a.k.a *normalization by evaluation*) and *common subexpression elimination* into an elegant optimization algorithm overcoming some challenges usually associated to partial evaluation.

In particular, partial evaluation of let-binding is not *safe* in the sense of always resulting in a better, or at least equal, performance than the original. This is because a variable may appear multiple times, hence substituting it multiple times would duplicate code. Ideally, common subexpression elimination (CSE) would remove redundancies introduced by partial evaluation. But the presence of scopes, as e.g., introduced by functions, loops, and branches – all constructs that prevail in array computations – can complicate CSE. While compilers can rectify these issues by using additional rules often summarized under the general term of *code motion*, this comes at the cost of having to decide in what order and how often to apply these additional rules, i.e., it implies creating an optimization schedule, which complicates the algorithm.

¹ We distinguish run-time (as in run-time library) from running time as the time it takes to run something.

<code>let z = let y1 = x + 1 2 * y1 let y2 = x + 1 ...</code>	<code>let y1 = x + 1 let z = 2 * y1 let y2 = x + 1 ...</code>	<code>let f = fun i:nat. let y1 = x + 1 2 * y1 let y2 = x + 1 ...</code>	<code>let z = if c then let y1 = x + 1 2 * y1 else 2 * x let y2 = x + 1 ...</code>
--	--	---	--

(a) Nested lets.

(b) Flat let-bindings.

(c) Functions.

(d) Branches.

Figure 1 Sample programs.

Instead of these complications, with $A_i\text{NF}$, we propose a novel intermediate representation for array programs based on logical foundations that avoids the complexity of optimization schedules. Like ANF, which is normal with regard to commuting conversion of let-bindings implying maximal flatness, $A_i\text{NF}$ is normal with regard to commuting conversions of for-loops, thus enabling what we call maximal loop fission. Maximal loop fission is a fundamental property to enable further optimizations such as dead code elimination or common subexpression elimination. We provide a translation of POLARA into $A_i\text{NF}$, which performs maximal loop fission and loop invariant code motion.

Contributions. In summary, this paper makes the following contributions:

- We present $A_i\text{NF}$, an intermediate representation that makes use of the unconventional idea of treating arrays as positive types from polarized type theory. $A_i\text{NF}$ is normal with regard to commuting conversion for both let-bindings and for-loops, leading to flat and maximally fissioned terms.
- We present POLARA, a simple surface array language, along with a translation of POLARA to $A_i\text{NF}$, for which we prove termination, type preservation, and maximal fission.
- We present an optimization algorithm for $A_i\text{NF}$ based on normalization by evaluation and common subexpression elimination, for which we prove termination and type preservation.

2 Problem Statement

Typed partial evaluation is a powerful optimization technique [17], which can reduce excessive terms by applying computation laws. For example, it can reduce a projection on a pair $(a, b).1 \equiv a$ by applying β -reduction. Or, it can eliminate superfluous branches like in $\text{if } x \text{ then } (\text{if } x \text{ then } a \text{ else } b) \text{ else } c \equiv \text{if } x \text{ then } a \text{ else } c$ by applying uniqueness laws (e.g., η -expansion).

But, while shining on its logical foundation, partial evaluation is not a *safe* optimization. A *safe* optimization has to either reduce the running time of a program or at least preserve it. Partial evaluation of let-bindings is not *safe* because a variable may appear multiple times, hence substituting it multiple times would duplicate code. Ideally, common subexpression elimination (CSE) would remove all redundancies introduced by partial evaluation. But scopes introduced by nested let-bindings, functions, and branches complicate CSE. For illustration, below we consider a few examples of redundancies that can occur in programs and how CSE handles them.

In Figure 1a, a program with nested let-bindings is shown. Here, the variable z is bound to $2 * y_1$, where y_1 is bound to the successor of x ; and then the variable y_2 is bound to the successor of x as well. It is easy to see that y_1 is redundant with y_2 , yet y_1 is not in scope at the definition of y_2 , so we cannot simply replace one by the other. The problem can be

avoided by bringing the program into a form, where no let-binding is nested inside another let-binding, such that all previously bound variables are in scope for the whole remaining expression. Consider the program shown in Figure 1b, which is equivalent to the previous program, but this time no expression has a subexpression. Now, the former definition is in scope at the latter definition, and thus y_2 can be replaced by y_1 , thereby eliminating a duplicate subexpression.

As mentioned, functions and branches introduce scope as well, and therefore complicate CSE. Yet, the solution of flattening the code is not as straightforward to apply. To illustrate the problem with functions, consider the program shown in Figure 1c, which defines a function f . Inside the function the successor of x is bound to y_1 , and outside the function it is bound redundantly to y_2 . To share the expressions, we could consider moving the definition of y_1 out of the functions. But moving an expression out of a function is not *safe*, as long as we do not know whether the function will be called at all.

To illustrate the problem with branches, consider the program shown in Figure 1d. Here, the result of a conditional expression is bound to the variable z , in one branch the successor of x is bound to y_1 , and after the conditional expression the successor of x is bound to the variable y_2 . Similar to the function case, to share the expressions, we could consider moving the definition of y_1 out of the branch, but that is again not a *safe* optimization, as long as we do not know that this branch is taken.

To rectify the issues outlined above compilers use additional rules often summarized under the general term of *code motion*. But this comes at the cost of introducing the problem of having to decide in what order and how often to apply these additional rules (i.e., creating an optimization schedule).

Our work avoids the complexity of optimization schedules. We argue that – instead of complicating the CSE algorithm with optimization schedules – a better approach is to design an intermediate representation for array programs, which like ANF bans nested expressions. This simplifies the optimization of array programs, which now can safely rely on algorithms based on logical foundations such as partial evaluation and CSE. The novel intermediate representation, called A_iNF , is informally presented in the following along with a simple surface arrays language and the optimized translation of the latter to the former.

3 A_iNF , Polara, and Simplified Optimizations

We describe the two key insights on which our approach is based (Sections 3.1 and 3.2), introduce POLARA and A_iNF by example (Section 3.3), and explain how A_iNF simplifies optimizations (Section 3.4).

3.1 The Duality of Functions and Arrays

The list type is an inductive datatype defined by its constructors `nil` for the empty list and `cons` for constructing a list from another list with an additional element. Accordingly, algorithms over lists work by recursion, expressed with functions and branches. As element access on lists is implemented by recursion, it has to traverse the list until the element is found, giving it running time linear in the size of the list. Arrays, on the other hand, enjoy constant-time access and feature bulk operations. The consequence is that arrays do not fit into the usual pattern of inductive data types. Nevertheless, because custom semantics would require further proofs to ensure soundness, arrays are occasionally modelled as lists, with the hint that the actual running time can differ (in Lean for example²).

² <https://lean-lang.org/lean4/doc/array.html>

In functional array languages, we exploit the equivalence of an array of type X and length n with a function from a natural number below n to a value of type X . Forward, this equivalence allow us to access (`get`) elements of an array by its index. Backward, we create (`tabulate`) an array from a function describing each individual element based on their index. The forward direction is indeed already very much ingrained in everyday programming, as array access `a[i]` and function application `a(i)` look very much alike in many languages, and even share identical syntax in some.

$$\text{Array}_n X \leftrightarrow (\text{Fin}_n \rightarrow X)$$

$$\begin{aligned} \text{get} : \text{Array}_n X &\rightarrow (\text{Fin}_n \rightarrow X) \\ \text{tabulate} : (\text{Fin}_n \rightarrow X) &\rightarrow \text{Array}_n X \end{aligned}$$

But something important changes in the conversion from a function to an array, and vice versa. Functions are *lazy*, in the sense that the evaluation of a function is delayed until it is applied, while arrays are *eager*, in the sense that all elements of an array have already been evaluated and on array access only need to be looked up. Also, in a language with (side) effects, the two types can be distinguished in that a function application can trigger effects, while an array access cannot trigger effects. Dually, constructing a function cannot trigger effects, while constructing an array can trigger effects.

We can put the relationship between functions and arrays on a logical foundation by considering the difference between positive and negative types [39]. A positive type is defined by a set of constructors (introduction forms), and we get a single corresponding destructor (elimination form) for it with one continuation for the content of each possible constructor (pattern matching). A negative type is defined by a set of destructors, and we get a single corresponding constructor for it that has to provide one value for each destructor to extract (copattern matching). Positive types are usually associated with eager (call-by-value) evaluation, and negative types with lazy (call-by-name) evaluation. Many types can be defined either as a positive or as a negative type. For illustration, we consider the positive and the negative formulations of the product type below.

Products as Positive and as Negative Types. The product type as a positive type \times has a single constructor (a, b) (INTRO). A corresponding destructor (ELIM) can be systematically derived as pattern matching on the constructor. Reduction (BETA) occurs when a destructor is applied to a constructor, and they eliminate each other.

$$\begin{array}{c} \text{INTRO} \\ \Gamma \vdash a : A \quad \Gamma \vdash b : B \\ \hline \Gamma \vdash (a, b) : A \times B \end{array} \qquad \begin{array}{c} \text{ELIM} \\ \Gamma \vdash p : A \times B \quad \Gamma, a : A, b : B \vdash c : C \\ \hline \Gamma \vdash \text{let } (a, b) = p; c : C \end{array}$$

$$\begin{array}{c} \text{BETA} \\ \Gamma \vdash a : A \quad \Gamma \vdash b : B \quad \Gamma, x : A, y : B \vdash c : C \\ \hline \Gamma \vdash (\text{let } (x, y) = (a, b); c) \equiv c[x := a, y := b] \end{array}$$

Alternatively, products can also be defined as negatives types \otimes . In this case, we give primacy to a set of destructors, namely the projections `p.fst` and `p.snd` (ELIM1, ELIM) to access the individual elements of a tuple p , and derive systematically the corresponding constructor (INTRO) providing one value for each destructor to extract. Beta reduction occurs (BETA1, BETA1) when a destructor is applied to a constructor by extracting the corresponding value.

$$\begin{array}{c}
 \text{ELIM1} \quad \frac{\Gamma \vdash p : A \otimes B}{\Gamma \vdash p.\text{fst} : A} \quad \text{ELIM2} \quad \frac{\Gamma \vdash p : A \otimes B}{\Gamma \vdash p.\text{snd} : B} \quad \text{INTRO} \quad \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash (\text{fst} = a; \text{snd} = b) : A \otimes B} \\
 \\
 \text{BETA1} \quad \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash (\text{fst} = a, \text{snd} = b).\text{fst} = a} \quad \text{BETA2} \quad \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash (\text{fst} = a, \text{snd} = b).\text{snd} = b}
 \end{array}$$

Functions as Negative and Positive Types. Usually, the function type is considered a negative type. It has a single destructor – function application $f a$ (ELIM) – and the corresponding constructor is systematically derived by copattern matching on the possible destructors (INTRO). When a destructor is applied to the constructor, we extract the value provided as the body of the function, and substitute the variable with the argument (BETA).

$$\frac{\text{ELIM} \quad \Gamma \vdash f : A \rightarrow B \quad \Gamma \vdash a : A}{\Gamma \vdash f a : B} \quad \frac{\text{INTRO} \quad \Gamma, a : A \vdash b : B}{\Gamma \vdash \text{fun } a. b : A \rightarrow B} \quad \frac{\text{BETA} \quad \Gamma, x : A \vdash b : B \quad \Gamma \vdash a : A}{\Gamma \vdash (\text{fun } x. b) a \equiv b[x := a]}$$

The function type can also be represented as a positive type. In this case, the function is primarily defined through its constructor, and the destructor is systematically derived from pattern matching on the constructor. But the interpretation of positive function types comes with some challenges for the metatheory. The introduction form of a function turns a term-in-the-context-of-a-variable $a : A \vdash b : B$ into a function $(\text{fun } a. b) : A \rightarrow B$. Thus, the corresponding elimination form of a function $(\text{fun } a. b) : A \rightarrow B$ should introduce a variable of type term-in-the-context-of-a-variable $a : A \vdash b : B$ into the context. But to properly model that, we need a judgment where we have a context in the context, in other words a “higher-order judgment” [26, 25]. A judgment is higher-order when an entailment \vdash occurs inside the context of another entailment. An implementation of higher-order judgments needs to ensure that a variable which has such a judgment as a type is only used in larger contexts, where all required variables are available. For example, $b : (a : A \vdash B) \vdash b : B$ is invalid, given that b must occur in a context where an $a : A$ is available; while $b : (a : A \vdash B) \vdash (\text{fun } a. b) : A \rightarrow B$ is valid, because a variable $a : A$ has been introduced such that the use of b afterwards is safe.

$$\begin{array}{c}
 \text{INTRO} \quad \frac{\Gamma, a : A \vdash b : B}{\Gamma \vdash (\text{fun } a. b) : A \rightarrow B} \quad \text{ELIM} \quad \frac{\Gamma \vdash f : A \rightarrow B \quad \Gamma, x : (a : A \vdash B) \vdash c : C}{\Gamma \vdash \text{let } (\text{fun } a. x) = f; c : C} \\
 \\
 \text{BETA} \quad \frac{\Gamma, a : A \vdash b : B \quad \Gamma, x : (a : A \vdash B) \vdash c : C}{\Gamma \vdash \text{let } (\text{fun } a. x) = (\text{fun } a. b); c \equiv c[x := b]}
 \end{array}$$

Interpreting positive function types as arrays. We avoid the challenges of interpreting functions as positive types by proposing to interpret positive function types as arrays, re-interpreting the rules of the positive function type as the rules of the array type. We require the argument type to be the type of natural numbers below some number n , which corresponds to the index type of an array. Traditionally, functional programming works with lists and not arrays, therefore the constant-time access of arrays is not accurately represented by the model; in functional array languages [15, 37], arrays tend to live in the shadow of the

function type, as their introduction and elimination forms depend on (higher-order) functions. Interpreting the array as a positive function type makes them independent and puts them on an equal footing to the other types with regard to their logical foundation.

We distinguish positive functions, i.e., arrays, from normal functions by using \Rightarrow for the type, writing $(\text{for } a. b)$ as the introduction form for arrays, while the elimination form is given, as always, by pattern matching on all possible introduction forms:

$$\begin{array}{c} \text{INTRO} \\ \frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash (\text{for } x. b) : A \Rightarrow B} A = \text{Fin}_n \end{array} \quad \begin{array}{c} \text{ELIM} \\ \frac{\Gamma \vdash f : A \Rightarrow B \quad \Gamma, x : (a : A \vdash B) \vdash c : C}{\Gamma \vdash \text{let } (\text{for } a. x) = f; c : C} A = \text{Fin}_n \end{array}$$

$$\begin{array}{c} \text{BETA} \\ \frac{\Gamma, a : A \vdash b : B \quad \Gamma, x : (a : A \vdash B) \vdash c : C}{\Gamma \vdash \text{let } (\text{for } a. x) = (\text{for } a. b); c \equiv c[x := b]} \end{array}$$

Intuitively, analogously to how pattern matching on a product introduces two variables (one for each projection of the product), pattern matching on an array introduces a family of variables, one for each element. For illustration, consider $b[a := 2]$ as b_2 , and $(\text{let } (\text{for } a. b) = f; c)$ as $(\text{let } (b_0, b_1, \dots, b_{n-1}) = f; c)$.

Arrays Enable CSE. Flattening let-bindings usually helps CSE. More precisely the rule that is used to create the ANF representation is the let-let commuting conversion:

$$\begin{array}{l} (\text{let } y = (\text{let } x = e1; e2); e3) \equiv \\ (\text{let } x = e1; \text{let } y = e2; e3) \end{array}$$

Note that the following let-fun commuting conversion is not *safe* because on the left-hand side $e1$ is evaluated at most once, even if it was used multiple times in $e2$; but on the right-hand side it will be evaluated once for each usage in $e2$.

$$\begin{array}{l} (\text{let } y = (\text{fun } i. \text{let } x = e1; e2); e3) \equiv \\ (\text{let } (\text{fun } i. x) = (\text{fun } i. e1); \text{let } y = (\text{fun } i. e2); e3) \end{array}$$

On the other hand, the let-for commuting conversion that we use in $A_i\text{NF}$ below is *safe* and states that the following two lines are equivalent. As array construction is evaluated eagerly, the expression $e1$ is evaluated just once for each iteration, on both sides of the equation.

$$\begin{array}{l} (\text{let } y = (\text{for } i. \text{let } x = e1; e2); e3) \equiv \\ (\text{let } (\text{for } i. x) = (\text{for } i. e1); \text{let } y = (\text{for } i. e2); e3) \end{array}$$

Intuitively, this rule allows us to split a complex loop into multiple simpler loops, hence it is closely connected to loop *fission*. If we use this rule to split every loop as much as possible, then we end up with a normal form in which every loop only contains a single operation. First performing loop fission as much as possible helps with implementing other optimizations, for example allows CSE to remove redundancies that it could not otherwise eliminate.

The use of this rule means that frequently both the left side and the right side of a variable definition are surrounded by the same form (on the left as a pattern form, on the right as a term form), so we will introduce some syntactic sugar and write $\text{let for } i. (x = e1); e2$ to mean $\text{let } (\text{for } i. x) = (\text{for } i. e1); e2$ in the following.

3.2 Lifting Branching into the Context

A different problem arises with values of the Boolean type, Booleans have two constructors, `true` and `false` (INTRO1, INTRO2). They have one destructor, the conditional expression (ELIM), where one continuation is provided for each constructor, the consequent and the alternative. A conditional reduces to the consequent when the condition is true (BETA1), and to the alternative when the condition is false (BETA2).

$$\begin{array}{c}
 \text{INTRO1} \qquad \qquad \text{INTRO2} \qquad \qquad \text{ELIM} \\
 \frac{}{\Gamma \vdash \text{true} : \text{bool}} \qquad \frac{}{\Gamma \vdash \text{false} : \text{bool}} \qquad \frac{\Gamma \vdash p : \text{bool} \quad \Gamma \vdash e : C \quad \Gamma \vdash f : C}{\Gamma \vdash \text{if } p \text{ then } e \text{ else } f} \\
 \\
 \text{BETA1} \qquad \qquad \qquad \text{BETA2} \\
 \frac{\Gamma \vdash e : C \quad \Gamma \vdash f : C}{\Gamma \vdash \text{if true then } e \text{ else } f \equiv e} \qquad \frac{\Gamma \vdash e : C \quad \Gamma \vdash f : C}{\Gamma \vdash \text{if false then } e \text{ else } f \equiv f}
 \end{array}$$

The let-if commuting conversion below is *safe* with regard to running time. But applying the commuting conversion duplicates the expression `e3`. If `e3` is a big expression, then even if duplicating it in different branches may not impact the running time, having nested branches will lead to a blow-up of the code size exponential in the number of branches (which is also bad for the compiling time).

```
(let z = (if e0 then e1 else e2); e3) ≡
(if e0 then (let z=e1; e3) else (let z=e2; e3))
```

Essentially, the above rule bubbles up conditionals to the top of the expression. Instead, we propose to trickle down the conditionals using conditionally defined variables and state a new `let-if` commuting conversion that does not duplicate the branches. To avoid duplicating branches, we introduce a syntactically single-branch `if`. A single-branch `if` produces a conditional value, i.e., a value that can only be accessed if the condition is true, and the single-branch `if!` produces a value that can only be accessed if the condition is false.

Analogously, we have let-bindings for conditional variables `let (if e. x) := ...` (or `let (if! e. x) := ...`), which define a variable `x` that can only be accessed if the condition `e` is true (or false, respectively). Two simple syntactical conditions can be used to check whether a conditional variable is accessible: First, a conditional variable is accessible on the right-hand side of the definition of another conditional variable that has the same condition. In other words, a conditional variable can be used to define the value of another conditional variable with the same condition. Second, a conditional variable is accessible in one of the branches of a standard two-branched `if` condition. In other words, two mutually exclusive conditional variables can be combined with an `if` to define a non-conditional variable.

Using the single-branch `if`, we can now express a `let-if` commuting conversion, that does not duplicate `e3`. Here the double-branched `if` is separated into two single-branch `ifs` and `e3` remains to be executed once afterwards:

```
(let z = (if e0 then e1 else e2);
e3)
≡
(let (if e0. z1) = (if e0. e1);
let (if! e0. z2) = (if! e0. e2);
let z = (if e0 then z1 else z2);
e3)
```

Table 1 Common linear algebra operations in POLARA; NumPy for reference.

Name	NumPy	POLARA
Vector addition	<code>v + w</code>	<code>for i. v[i] + w[i]</code>
Matrix addition	<code>A + B</code>	<code>for i j. A[i,j] + B[i,j]</code>
Element-wise product (vector)	<code>v * w</code>	<code>for i. v[i] * w[i]</code>
Element-wise product (matrix)	<code>A * B</code>	<code>for i j. A[i,j] * B[i,j]</code>
Outer product	<code>np.multiply.outer(A, B)</code>	<code>for i j k l. A[i,j] * B[k,l]</code>
Trace	<code>A.trace()</code>	<code>sum i. A[i, i]</code>
Transpose	<code>A.transpose()</code>	<code>for i j. A[j, i]</code>
Matrix multiplication	<code>A @ B</code>	<code>for i k. sum j. A[i,j] * A[j,k]</code>
Matrix-vector multiplication	<code>A @ v</code>	<code>for i. sum j. A[i,j] * v[j]</code>

Correct use of conditional values will thus frequently lead to the use of the same condition on the variable bound by a `let` and in a single-branched if in the bound expression. We will thus make use of syntactic sugar writing `let if e0. (x1 = e1); e2` to mean `let (if e0. x1) = (if e0. e1); e2`.

3.3 Polara and A_iNF by Example

In this section, we informally introduce both POLARA and A_iNF . We do so by giving examples of array operations and programs in POLARA and showing the result of compiling them to A_iNF . Just for reference, we will also provide versions of the POLARA examples written in the widely-adopted array programming library NumPy [14]. Please note that the focus of this paper lies in the exploration of A_iNF , rather than POLARA. The latter serves merely as a vehicle to elucidate how A_iNF effectively facilitates optimizations during the translation process from a surface array language. Hence, compared to NumPy, we have purposely kept it closer to low-level imperative code.

In POLARA, an expression e is either a constant c , or an arithmetic operator $e + e$, function application $e e$, array access $a[i]$, array construction `for i:n. e`, or summation `sum i:n. e`, as well as pairs (e, e) and projection $e.1, e.2$. The array construction `for i:n. e` constructs an array of length n by repeatedly evaluating e , with i bound to the values from 0 to $n-1$. For example, `for i:3. 10*i` evaluates to `[0, 10, 20]`. We will write `for i. e` if the size of the array can be inferred from the context. Summation is syntactic sugar for constructing an array and then summing it, so `sum i.e ≡ sum (for i. e)`.

In Table 1, we list several common linear algebra operations and compare how they can be expressed using the linear algebra library NumPy and POLARA. We assume as given that the vectors v , w and matrices A , B are of appropriate sizes.

Dense Layer. As a slightly more involved example, we show how a dense neural network layer can be implemented in NumPy, POLARA, and A_iNF , respectively. The NumPy example makes use of the built-in matrix multiplication operator \circledast . While such an operation can be implemented as function in POLARA, we show an example that only relies on the few POLARA primitives, using the `for` looping construct and indexing. Likewise, while the NumPy definition uses the built-in `maximum` function and addition, the POLARA version uses an explicit loop that performs element-wise multiplication and additions across the vectors.

Compared to the untyped NumPy program, we also declare the types of the arguments. A type $n \Rightarrow \text{flt}$ describes an array of floating point numbers with size n .

Obviously, the corresponding A_iNF program (Figure 2a) is rather lengthy, as every intermediate result gets assigned to a variable, just like in ANF.

33:10 Compiling with Arrays

<pre>def dense(b, W, x): return np.maximum(0, W @ x + b)</pre>	<pre>dense(b:n⇒flt, W:n⇒m⇒flt, x:m⇒flt): n⇒flt := for i. max(0, (sum j. W[i][j] * x[j]) + b[i])</pre>
NumPy	POLARA

Convolution. We now describe how to express convolution in POLARA. Convolution involves moving a vector, called the kernel, across another vector while repeatedly calculating the dot product. For this example, we need to subtract two indices to indicate that we shift one array while keeping the other as it is. In the A_iNF example (Figure 2b), we create a two-dimensional array $x10$ containing all the possibilities for shifting the array y . For example if $y = [1,2,3]$, then $x10$ is a matrix of size 3×3 so that $\text{tmp1} = [[1,2,3], [2,3,1], [3,1,2]]$. We then form the dot product of each entry with x .

<pre>def conv(x, y): return np.convolve(x, y, 'same')</pre>	<pre>conv(x: n⇒flt, y: (n+m-1)⇒flt): m⇒flt := for i. sum j. x[j] * y[j+i]</pre>
NumPy	POLARA

Black-Scholes. Black-Scholes is a simplified mathematical model for the dynamics of derivative investments in financial markets. The Black-Scholes formula provides an estimate for the price of the call option (buying) and the put option (selling) of a European-style option given the original price S , the strike price K , the expiration time T , the force-of-risk r and the standard deviation σ . The interesting part, from an array programming language's perspective, is that with a naive implementation of the calls and puts as separate functions, common subexpression elimination is not able to identify the redundant computation across these functions over two separate loops.

In particular, note the redundant definition of $d1$ and $d2$ in the `calls` and the `puts` function. This code gets inlined into the `blackScholes` function, but the two function calls land in separate loops. Nevertheless, using loop fission the output can be reduced to just 22 lines of A_iNF (see Figure 2c); without fission and CSE the generated code would have 54 lines.

<pre>calls(S: flt, K: flt, T: flt: r: flt: sigma: flt): flt := let d1 := (log (S / K) + (r + sigma * sigma / 2) * T) / (sigma * sqrt T) let d2 := d1 - sigma * sqrt T S * normCdf d1 - K * exp (0 - var r * var T) * normCdf d2 puts(S: flt, K: flt, T: flt: r: flt: sigma: flt): flt := let d1 := (log (S / K) + (r + sigma * sigma / 2) * T) / (sigma * sqrt T) let d2 := d1 - sigma * sqrt T K * exp (0 - r * T) * normCdf (0 - d2) - S * normCdf (0 - d1) blackScholes(arr: (n ⇒ flt)): n ⇒ (flt × flt) := let S := 1; let K := 1; let r := 1; let sigma := 1 let Calls: (n ⇒ flt) := for i. calls(S, K, arr[i], r, sigma) let Puts: (n ⇒ flt) := for i. puts(S, K, arr[i], r, sigma) for i. (Calls[i], Puts[i])</pre>	POLARA
--	--------

3.4 Simplifying Optimizations with A_iNF

This section provides an overview showing how some classical optimizations can be applied to A_iNF and illustrates why our novel normal form simplifies their implementation.

To improve readability, we will sometimes present A_iNF code in a way that deviates from the actual representation by putting multiple operations in one line, when this does not affect the optimization.

```

dense(b: n => flt, W: n => m => flt,
      x: m => flt): n => flt :=
let for i:n, (x0 : flt := 0)
let for i:n, j:m, (x1 : m => flt := W[i])
let for i:n, j:m, (x2 : flt := x1[j])
let for i:n, j:m, (x3 : flt := x[j])
let for i:n, j:m, (x4 : flt := x2 * x3)
let for i:n, (x5 : m => flt := for j:m, x4)
let for i:n, (x6 : flt := sum x5)
let for i:n, (x7 : flt := b[i])
let for i:n, (x8 : flt := x6 + x7)
let for i:n, (x9 : flt := max x0 x8)
let (x10 : n => flt := for i:n, x9)
x10

```

A_iNF

(a) A_iNF for a dense layer.

```

conv(x: n => flt, y: p => flt): m => flt :=
let for i:m, j:n, (x0 : flt := x[j])
let for i:m, j:n, (x1 : fin p := j + i)
let for i:m, j:n, (x2 : flt := y[x1])
let for i:m, j:n, (x3 : flt := x0 * x2)
let for i:m, (x4 : n => flt := for j:n, x3)
let for i:m, (x5 : flt := sum x4)
let (x6 : m => flt := for i:m. x5)
x6

```

A_iNF

(b) A_iNF for convolution.

```

blackScholes(arr: n => flt): n => flt * flt :=
let for i1:n, (x0 : flt := 1.500000)
let for i1:n, (x1 : flt := i0[i1])
let for i1:n, (x2 : flt := x0 * x1)
let for i1:n, (x4 : flt := sqrt x1)
let for i1:n, (x5 : flt := x2 / x4)
let for i1:n, (x6 : flt := normCdf x5)
let for i1:n, (x7 : flt := 0.000000)
let for i1:n, (x9 : flt := x7 - x1)
let for i1:n, (x10 : flt := exp x9)
let for i1:n, (x19 : flt := x5 - x4)
let for i1:n, (x20 : flt := normCdf x19)
let for i1:n, (x21 : flt := x10 * x20)
let for i1:n, (x22 : flt := x6 - x21)
let for i1:n, (x37 : flt := x7 - x19)
let for i1:n, (x38 : flt := normCdf x37)
let for i1:n, (x39 : flt := x10 * x38)
let for i1:n, (x47 : flt := x7 - x5)
let for i1:n, (x48 : flt := normCdf x47)
let for i1:n, (x49 : flt := x39 - x48)
let for i1:n, (x50 : (flt * flt) := (x22, x49))
let (x51 : (n => flt * flt) := for i1:1, x50)
x51

```

A_iNF

(c) A_iNF for a Black-Scholes.**Figure 2** Generated A_iNF.

Loop Fission. Loop fission is an optimization pass that prepares code to improve the effectiveness of dead code elimination. Standard dead code elimination can only delete whole loops. Loop fission splits a loop into parts, so that individual parts that are not used can be removed. In A_iNF, the body of each loop is a single operation, which means that any A_iNF program is necessarily as fissioned as possible. A simple partial evaluation pass on A_iNF can then perform dead-code elimination.

Below, the left side shows an array computation in POLARA. On the right, that same computation has been transformed to A_iNF, which implies loop fission. The code follows the principle from ANF that expressions should be atomic, i.e., only have one operation. In terms of array programming, this leads to the first loop being split into three loops. Partial evaluation could then reduce the full program to just `for i. f(xs[i])`.

```

let x = for i.
  let ys = f(xs[i])
  let zs = f(xs[i])
  (ys, zs)
for i.
  fst x[i]

```

POLARA

```

let for i. (ys = f(xs[i]))
let for i. (zs = f(xs[i]))
let for i. (x = (ys, zs))
let for i. (y = fst x)
let z = for i. y
z

```

A_iNF

A further advantage of loop fission is that it improves loop *fusion*: Splitting a program into as many loops as possible, gives more freedom to the algorithm for combining loops again.

33:12 Compiling with Arrays

	$n \in \mathbb{N}$	$f \in \mathbb{F}$	$x \in \text{Var}$	$i \in \text{Idx}$
Types	$t ::= \text{fin } n \mid \text{flt} \mid t \hat{\times} t \mid t \hat{\rightarrow} t \mid n \Rightarrow t$			
Constants	$c ::= n \mid f \mid \hat{+} \mid \hat{:} \mid \hat{-} \mid \hat{/} \mid \text{app} \mid \text{get} \mid \text{pair} \mid \text{fst} \mid \text{snd} \mid \text{sum}$			
POLARA	$e ::= c \bar{e} \mid x \mid \text{fun } x:t. e \mid \text{for } x:n. e \mid \text{ite } e \ e \ e \mid \text{let } e; \ e$			
$A_i\text{NF}$	$a ::= \text{let } C[x = p]; \ a \mid x$			
Primitives	$p ::= c \bar{x} \mid i \mid \text{fun } i:t. x \mid \text{for } i:n. x \mid \text{ite } x \ x \ x$			
Scope Contexts	$C[\cdot] ::= \cdot \mid C[\text{fun } i:t. \cdot] \mid C[\text{for } i:n. \cdot] \mid C[\text{if } x \neq 0. \cdot] \mid C[\text{if } x = 0. \cdot]$			

Figure 3 POLARA and $A_i\text{NF}$.

Common subexpression elimination. We can now see how $A_i\text{NF}$ helps with CSE. On the left, we recapitulate the example from above; on the right, we can see the same program in $A_i\text{NF}$.

<pre> let f = for i. let y = x+1 let y' = 2*y y' let z = x+1 ... </pre>	<hr/> POLARA	<pre> let for i. (y = x + 1) let for i. (y' = 2 * y) let for i. (f = y') let for i. (z = x + 1) ... </pre>	<hr/> $A_i\text{NF}$
---	---	---	--

The loop computing f has been broken down into two loops. As a result, z is clearly redundant, as it performs the same computation in the same scope as y . Therefore, compared to array languages using higher-order functions, $A_i\text{NF}$ allows us to use the simple, standard approach to CSE, and nonetheless remove redundancies between expressions inside and outside of loops.

Loop invariant code motion. Loop invariant code motion (LICM), which moves constants out of a loop, is another optimization that benefits from $A_i\text{NF}$. In $A_i\text{NF}$, this would correspond to dropping an unused index; hence, the implementation of LICM is very simple. On the left, we generate an array ys , in which every element is the constant 1 . We then compute an array zs that makes use of ys . Notice that the index i that is bound in the creation of ys is not used. We can therefore eliminate that loop, adjusting uses of ys accordingly from $\text{ys}[i:=j]$ to ys , as seen on the right.

<pre> let for i. (ys = 1) let zs = for i. f(xs[i], ys) ... </pre>	<hr/> $A_i\text{NF}$	<pre> let ys = 1 let zs = for i. f(xs[i], ys) ... </pre>	<hr/> $A_i\text{NF}$
---	--	--	--

4 Mechanization

We mechanized POLARA, partial evaluation of POLARA, $A_i\text{NF}$, the translation from POLARA to $A_i\text{NF}$, and common subexpression elimination over $A_i\text{NF}$, using the dependently typed programming language Lean 4 [7].

4.1 Polara and Partial Evaluation

Polara. The POLARA grammar uses the set of natural numbers, floating point numbers, variables and indices (Figure 3), but the distinction between variables and indices is only relevant for $A_i\text{NF}$. Types are floating point numbers, products, functions, and arrays, as well as bounded natural numbers, i.e. $\text{fin } n$ is the type of numbers smaller than n . Constants are

$$\begin{array}{c}
\frac{n < m}{\vdash n : \text{fin } m} \quad \vdash f : \text{flt} \quad \vdash \text{app} : (t_1 \hat{\rightarrow} t_2) \rightarrow t_1 \rightarrow t_2 \quad \vdash \text{get} : (n \Rightarrow t_1) \rightarrow \text{fin } n \rightarrow t_1 \\
\vdash \text{pair} : t_1 \rightarrow t_2 \rightarrow (t_1 \hat{\times} t_2) \quad \vdash \text{fst} : (t_1 \hat{\times} t_2) \rightarrow t_1 \quad \vdash \text{snd} : (t_1 \hat{\times} t_2) \rightarrow t_2 \\
\vdash \hat{+} : \text{fin } n \rightarrow \text{fin } m \rightarrow \text{fin } (n + m - 1) \quad \vdash \hat{+} : \text{flt} \rightarrow \text{flt} \rightarrow \text{flt} \quad \vdash \text{sum} : (n \Rightarrow \text{flt}) \rightarrow \text{flt} \\
\begin{array}{c}
\text{VAR} \qquad \text{CONST} \\
\frac{x:t \in \Gamma}{\Gamma \vdash x : t} \quad \frac{\vdash c : \bar{t}_i \rightarrow t' \quad \Gamma \vdash e_i : t_i}{\Gamma \vdash c \bar{e}_i : t'}
\end{array} \\
\begin{array}{c}
\text{FUN} \qquad \text{FOR} \\
\frac{\Gamma, x:t_1 \vdash e : t_2}{\Gamma \vdash \text{fun } x:t_1. e : t_1 \rightarrow t_2} \quad \frac{\Gamma, i:\text{fin } n \vdash e_2 : t}{\Gamma \vdash \text{for } i:n. e_2 : n \Rightarrow t}
\end{array} \\
\begin{array}{c}
\text{LET} \qquad \text{ITE} \\
\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma, x:t_1 \vdash e_2 : t_2}{\Gamma \vdash \text{let } x = e_1; e_2 : t_2} \quad \frac{\Gamma \vdash e_1 : \text{fin } 2 \quad \Gamma \vdash e_2 : t \quad \Gamma \vdash e_3 : t}{\Gamma \vdash \text{ite } e_1 e_2 e_3 : t}
\end{array}
\end{array}$$

■ **Figure 4** POLARA’s type system.

natural number and floating point literals, arithmetic symbols, function application (`app`), array access (`get`), pair construction (`pair`), first and second projection (`fst`, `snd`), and array summation (`sum`). POLARA terms are variable access, n-ary constant application, function abstraction, array construction, branching (`ite`), and let-binding. We decided to put first-order syntax forms such as function application, array access, pairing, and the product projections into the constants, because they are all handled uniformly by the following algorithms, while the higher-order syntax forms, i.e., the ones that bind variables, such as function abstraction, array construction, branching, and let-binding are kept in the terms because they are all treated differently.

Intrinsic Types. The typing rules for POLARA are given in Figure 4. First, we give the types for constants. Note that we use the \rightarrow symbol for the typing judgement of constants that take arguments. This is not to be confused with the type constructor $\hat{\rightarrow}$. The typing rules for variables (VAR), function abstractions (FUN), and `let`-bindings (LET) are standard. The rule CONST allows one to apply a constant to a number (possibly zero) of arguments. For example, as `app` has type $(t_1 \hat{\rightarrow} t_2) \rightarrow t_1 \rightarrow t_2$, the expression `app` $e_1 e_2$ has type t_2 when $e_1 : t_1 \hat{\rightarrow} t_2$ and $e_2 : t_1$. The FOR rule shows that constructing an array with `for` requires an expression of type `nat` for the size and another expression, which can use the (numerical) index i and whose type gives the element type of the array. The ITE rule states that the condition has to be of type `nat` and the two branches have to be of the same type (the condition is considered true if nonzero).

PHOAS. Our formal development uses parametric higher-order abstract syntax (PHOAS) [29, 5], allowing us to leverage the binders of the host language as binders for the guest language. Terms are parametrized by an abstract denotation of types Γ , and variables contain a value of that type. By using PHOAS, we can avoid certain technicalities relating to variable binding such as capture-avoiding substitution, thereby streamlining the implementation.

Static Size. As mentioned above, our array types have the form $n \Rightarrow a$, where n is the size of the array. The fact that the size of an array is always part of its static type, implies that the sizes of all arrays are known at compile time. This guarantees that indexing can be statically checked for out-of-bounds array accesses, ensuring the absence of run time errors without requiring run time checks.

Because POLARA is not polymorphic, expressions operating on arrays are fixed to specific array sizes. For example, there is no single expression in POLARA that can map a function over an array of *arbitrary* size. This restriction is alleviated because our language is embedded, allowing us to reuse polymorphism from the host language. More concretely, we can define a function in the host language that for each number n returns a POLARA term implementing `map` on an array of size n (here, λ belongs to the host language and `fun` belongs to POLARA):

```
map  :  (n : ℕ) → (Γ ⊢ (t1 ↗ t2) ↗ (n ⇒ t1) ↗ (n ⇒ t2))
map := λn. fun f a. for i. f a[i]
```

Termination. The use of static array sizes ensures that array indexing is total. In fact, every language construct in POLARA is deterministic and terminating, making the language total; hence it is not Turing-complete. Most notably, we eschew general recursion in favor of the more well-behaved looping construct `for`. The lack of non-termination allows us to give a simple denotational semantics and guarantees termination of normalization, as described next.

Normalization by Evaluation (NbE). Normalization is defined in Figure 5 by a denotation for types ($\llbracket t \rrbracket_\Gamma : \text{Type}$), a corresponding denotation for terms and constants (such that when e has type t , then $(\llbracket e \rrbracket_\Gamma : \llbracket t \rrbracket_\Gamma)$), as well as functions `quote` (η), `splice` (η'), and `norm`. Note the additional argument Γ – this is a peculiarity of the PHOAS representation, where Γ determines the denotation of variables. This argument can take different values, depending on which information we want to extract from a term. For example, when pretty-printing a term we want to produce a string, so we associate every variable also with a string ($\Gamma t := \text{String}$). For NbE, every term should be translated to the denotation of their type, so we associate every variable to the denotation $\llbracket \cdot \rrbracket_\Gamma$ of its type using Γ . The function `norm` takes a value of type $(\forall \Gamma. \Gamma \vdash t)$ and returns one of the same type. The quantification means that we can only use variables that were created by the language’s binding constructs, so the type represents closed terms.

The denotation of a bounded natural number is a bounded natural number term, the denotation of a floating point number is a floating point number term, the denotation of a product is a product of the denotations, the denotation of a function is a function of the denotations, the denotation of an array is a function from a bounded natural number term to a denotation of the array’s content. Later, for code generation, we will again distinguish functions and arrays. But for the purpose of normalization by partial evaluation (NbE), we model arrays as functions so as to reduce the need for rules for both of them.

The quote η and splice η' functions perform eta-expansion of terms by recursion over the types. Quote turns denotations into terms, and splice turns terms back into denotations. The denotation of a POLARA term is a corresponding host-language value of that term (i.e., a Lean value in our mechanization). NbE then evaluates terms in the environment of splicing, followed by quoting the denotation back into a term.

Constants denote functions that check for whether their argument is known, and the partial evaluation of their argument; otherwise, they quote/splice the term into a denotation of the type.

$$\begin{array}{lll}
 \llbracket \cdot \rrbracket_\Gamma & : \text{Ty} \rightarrow \text{Type} \\
 \llbracket \text{fin } n \rrbracket_\Gamma & = \Gamma \vdash \text{fin } n & \llbracket \text{ite } e_1 \ e_2 \ e_3 \rrbracket = \\
 \llbracket \text{flt} \rrbracket_\Gamma & = \Gamma \vdash \text{flt} & \\
 \llbracket t_1 \hat{\times} t_2 \rrbracket_\Gamma & = \llbracket t_1 \rrbracket_\Gamma \times \llbracket t_2 \rrbracket_\Gamma & \\
 \llbracket t_1 \hat{\Rightarrow} t_2 \rrbracket_\Gamma & = \llbracket t_1 \rrbracket_\Gamma \rightarrow \llbracket t_2 \rrbracket_\Gamma & \\
 \llbracket n \Rightarrow t \rrbracket_\Gamma & = (\Gamma \vdash \text{fin } n) \rightarrow \llbracket t \rrbracket_\Gamma & \begin{cases} \llbracket e_2 \rrbracket & \text{if } \llbracket e_1 \rrbracket = 1 \\ \llbracket e_3 \rrbracket & \text{if } \llbracket e_1 \rrbracket = 0 \\ \eta'(\text{ite } \llbracket e_1 \rrbracket (\eta \llbracket e_2 \rrbracket) (\eta \llbracket e_3 \rrbracket)) & \text{otherwise} \end{cases}
 \end{array}$$

(a) Denotation of types.

(b) Denotation of ite.

$$\begin{array}{llll}
 \llbracket \cdot \rrbracket & : ((\llbracket \cdot \rrbracket_\Gamma \vdash t) \rightarrow \llbracket t \rrbracket_\Gamma) & \eta & : \forall t. \llbracket t \rrbracket_\Gamma \rightarrow (\Gamma \vdash t) \\
 \llbracket x \rrbracket & = x & \eta (t_1 \hat{\Rightarrow} t_2) e & = \text{fun } i : t_1. \eta t_2 (e (\eta' t_1 i)) \\
 \llbracket \text{fun } i. e \rrbracket & = \lambda i. \llbracket e \rrbracket_i & \eta (n_1 \hat{\Rightarrow} t_2) e & = \text{for } i : n_1. \eta t_2 (e (\eta' t_1 i)) \\
 \llbracket \text{for } i. e \rrbracket & = \lambda i. \llbracket e \rrbracket_i & \eta (t_1 \hat{\times} t_2) e & = \text{tup } (\eta t_1 e.1) (\eta t_2 e.2) \\
 \llbracket c \bar{e} \rrbracket & = \llbracket c \rrbracket \llbracket \bar{e} \rrbracket & \eta (\text{fin } n) e & = e \\
 \llbracket \text{let } e_1; e_2 \rrbracket & = \llbracket e_2 \rrbracket \llbracket e_1 \rrbracket & \eta \text{ flt } e & = e \\
 \\
 \llbracket \text{app} \rrbracket e_1 e_2 & = e_1 e_2 & \eta' & : \forall t. (\Gamma \vdash t) \rightarrow \llbracket t \rrbracket_\Gamma \\
 \llbracket \text{get} \rrbracket e_1 e_2 & = e_1 e_2 & \eta' (t_1 \hat{\Rightarrow} t_2) e & = \lambda i. \text{app } (\eta' t_2 e) (\eta t_1 i) \\
 \llbracket \text{pair} \rrbracket e_1 e_2 & = (e_1, e_2) & \eta' (n_1 \hat{\Rightarrow} t_2) e & = \lambda i. \text{get } (\eta' t_2 e) (\eta t_1 i) \\
 \llbracket \text{fst} \rrbracket e & = e.1 & \eta' (t_1 \hat{\times} t_2) e & = (\eta' t_1 (\text{fst } e), \eta' t_2 (\text{snd } e)) \\
 \llbracket \text{snd} \rrbracket e & = e.2 & \eta' (\text{fin } n) e & = e \\
 \llbracket \text{sum} \rrbracket e & = \eta' (\text{sum } (\eta e)) & \eta' \text{ flt } e & = e \\
 \llbracket \hat{+} \rrbracket n_1 n_2 & = n_1 + n_2 & \text{norm} & : (\forall \Gamma. \Gamma \vdash t) \rightarrow (\forall \Gamma. \Gamma \vdash t) \\
 \llbracket \hat{+} \rrbracket e_1 e_2 & = e_1 \hat{+} e_2 & \text{norm } e & = \eta \llbracket e \rrbracket
 \end{array}$$

(c) Denotation of terms and constants.

(d) Quote η , splice η' , and normalization norm.**Figure 5** Typed partial evaluation.

4.2 A_iNF and Common Subexpression Elimination

FOAS. An essential component for implementing common subexpression elimination is the ability to compare to terms for equality. As we cannot decide equality over functions, we have to convert from parametric higher-order abstract syntax (PHOAS) to first-order syntax (FOAS) to get decidable equality for identifiers and terms containing variables.

A_iNF. In A_iNF, we distinguish between variables x and indices i (Figure 3). Variables are introduced by let-binding, while indices are introduced by functions and loops. An A_iNF term is a sequence of pattern-matching let-bindings of primitives, ending in a final variable (Figure 3, A_iNF). An essential property of A_iNF is thus, that it is both maximally fissioned (each for loop just has a single operation as a body) and maximally flat (an A_iNF term is a single list of terms without subterms, executed one after another). Pattern matching contexts C have one hole for the variable, and one form for each higher-order argument to any term former, namely array construction, function abstraction, if-consequence, and if-alternative. Primitives are constant application, indices, variable access, function abstraction, and array construction.

33:16 Compiling with Arrays

$C[\cdot]$	$: (\vdash t_1) \rightarrow (\text{Var } t_1 \rightarrow \text{A}_i\text{NF } t_2) \rightarrow \text{A}_i\text{NF } t_2$
$C[x]$	$= (\lambda x) k$
$C[i]$	$= (\lambda i) k$
$C[c e_1 e_2]$	$= C[e_1] \lambda x_1. C[e_2] \lambda x_2. (\lambda c x_1 x_2) k$
$C[\text{fun } i:t. e]$	$= C[\text{fun } i:t. \cdot] [e] \lambda x. (\lambda \text{fun } i:t. x) k$
$C[\text{for } i:e_1. e_2]$	$= C[e_1] \lambda x_1. C[\text{for } i:x_1. \cdot] [e_2] \lambda x_2. (\lambda \text{for } i:x_1. x_2) k$
$C[\text{ite } e_1 e_2 e_3]$	$= C[e_1] \lambda x_1. C[\text{if } x_1=0. \cdot] [e_2] \lambda x_2. C[\text{if } x_2 \neq 0. \cdot] [e_3] \lambda x_3. (\lambda \text{ite } x_1 x_2 x_3) k$
$C[\text{let } e_1; e_2]$	$= C[e_1] \lambda x_1. C[e_2 x_1] k$

(a) Fission.

$$\begin{aligned} (\cdot) &: \text{Prim } t_1 \rightarrow (\text{Var } t_1 \rightarrow \text{A}_i\text{NF } t_2) \rightarrow \text{A}_i\text{NF } t_2 \\ (\lambda x) k &= k x \\ (\lambda p) k &= \text{let } x = p; k x \quad \text{where } x \text{ unique} \end{aligned}$$

(b) Smart binding.

Figure 6 Fission with smart binding.

Conversion to A_iNF (Figure 6) exploits the fact that continuation-passing-style automatically flattens code. The function $C[e] k$ takes as inputs a POLARA term e , a pattern matching context C , and a continuation k , and returns an A_iNF term. In the mechanization the function uses a reader monad as well to generate unique variable names. The function is initialized with the empty pattern matching context, and the identity continuation; the variable counter is initialized with zero.

Another important helper function is smart binding $(\lambda p) k$, which takes a primitive p and a continuation k . Smart binding ensures that every primitive term passed to it is bound to a variable name, and that variable name is passed to the continuation. If the primitive term is a variable already, this variable name is passed to the continuation; otherwise the term is bound to a unique variable name, incrementing the counter.

Translation to A_iNF by $C[e] k$ recurses structurally over the term e . In the case of a variable or an index, the term is forwarded to smart binding. In the case of a constant application (exemplary shown for binary constant application), first the first subterm is translated, then in the continuation the second subterm is translated, and in the continuation the term is reconstructed as a primitive with variable referencing the name of the translated subterms, which is passed to smart binding to generate a new name for this term, passing the continuation along. In the case of function abstraction, array construction, and conditional expressions, the subterms are translated as well, but in adapted contexts, and the final term is passed as well to smart binding to generate a name for it, and the continuation is passed along. Concretely, in the case of function abstraction, the function body is translated in a context which includes the function argument. In the case of array construction, the array body is translated in a context which includes the iteration variable. In the case of conditional expressions, the consequence is translated in a context which includes the condition, and the alternative is translated in a context which includes the negation of the condition. Finally, in the case of let-binding, first the right-hand side of the binding is translated, and then the body of the binding.

CSE. In addition to deciding equality for terms, a further complication with common subexpression elimination is that we also need to decide equality in the presence of already established equalities. For example consider the term $x=v$, $y=v$, $z=(x+y)$, $q=(y+x)$, $t=z+1$,

$$\begin{aligned}
 \text{Ren} &= [(t_1 : \text{Ty}) \times \text{Var } t_1 \times \text{Var } t_1] \\
 \text{Nam} &= [(t_1 : \text{Ty}) \times \text{Prim } t_1 \times \text{Var } t_1] \\
 \text{CSE} &: \text{Ren} \rightarrow \text{Nam} \rightarrow \text{A}_i\text{NF } t_2 \rightarrow \text{A}_i\text{NF } t_2 \\
 \text{CSE } r \sigma x &= (\sigma, \text{ren}_r x) \\
 \text{CSE } r \sigma (\text{let } C[x = p]; a) &= \begin{cases} \text{CSE } r (\text{let } C'[x = p']; \sigma) a & \text{if } \text{lookup } \sigma C' x p' = \text{none} \\ \text{CSE } ([x := x'] :: r) \sigma' a & \text{if } \text{lookup } \sigma C' x p' = \text{some } (x', \sigma') \\ \text{where } p' = \text{ren}_r p \\ \text{where } C' = \text{ren}_r C \end{cases}
 \end{aligned}$$

■ **Figure 7** Common subexpression elimination.

`r=q+1,` Correct CSE should eliminate it to `x=v, z=(x+x); t=z+1; rename [y→x; q→z; r→t]`, Notice how the later eliminations are dependent on the earlier ones. If we simply rename the remaining term every time we detect a variable to be redundant, then this algorithm would perform exponentially worse, because every renaming is a traversal over the whole remaining term, and CSE itself is already a traversal over the whole term. To keep everything with a single traversal, we adapt CSE to carry a renaming with it, which is applied just before a term is checked for redundancy.

CSE (Figure 7) takes a renaming, a naming, and an A_iNF term, and returns a new A_iNF term of the same type. A renaming is a list of pairs of variables of the same type, representing that the first variable is to be replaced by the second. A naming is a list of pairs of a primitive and a variable of the same type, representing that the primitive term has been previously bound to that variable. CSE works by structural recursion over the term. When the input term is just a variable, it simply applies the renaming. When the input term is a let-binding, then the renaming is applied to the term as well. The renamed term is looked up in the list of previously defined terms. If the term has not been bound to a variable name already (none), then the term is now let-bound to a variable, inside a renamed pattern matching context C . CSE proceeds with the remaining terms a , remembering that the term p' has been bound to σ , so that future redundant occurrences of p' can be eliminated. If the term has already been bound to a variable name (some x'), then no let-binding is produced, but only the renaming is extended to replace future references to x to the already existing x' instead. CSE proceeds with the remaining terms a , remembering that the term p' has been bound to σ , so that future redundant occurrences of p' can be eliminated.

4.3 Mechanization in Lean

In this section, we present excerpts from the Lean mechanization and relate them to the paper formalization. The type of terms `Tm` corresponds to $(\Gamma \vdash t)$ and features constructors for variables and constants (`var`, `cst0` etc.). In the paper, we do not write these constructors explicitly, so we would write `x` rather than `var x`. We define the following types corresponding to the above definitions of syntax (Figure 3) in Lean.

33:18 Compiling with Arrays

```

inductive Var : Ty → Type -- Variables Var
inductive Par : Ty → Type -- Indices Iidx

inductive Ty           -- Types t
inductive Const0 : Ty → Type      -- Constants c (nullary)
inductive Const1 : Ty → Ty → Type -- Constants c (unary)
inductive Const2 : Ty → Ty → Ty → Type -- Constants c (ternary)
inductive Tm (Γ: Ty → Type): Ty → Type -- Terms e

inductive Prim : Ty → Type -- Primitives p
inductive Env : Type       -- Scoped Contexts C
inductive AINF : Ty → Type -- AINF a

```

Lean

In particular, we define the following functions in Lean. The function `Ty.de` corresponding to denotation of types $\llbracket \cdot \rrbracket_\Gamma$, `quote` to η and `splice` to η' , `Const0.de`, `Const1.de`, `Const2.de` and `Tm.de` were shown as term, constant, and its denotations $\llbracket \cdot \rrbracket$. Finally, `norm` is defined using `term` denotations and `quote`.

```

def Ty.de (Γ : Ty → Type): Ty → Type

def quote {Γ} : {α : Ty} → Ty.de Γ α → Tm Γ α
def splice {Γ} : {α : Ty} → Tm Γ α → Ty.de Γ α

def Const0.de : Const0 α → Ty.de Γ α
def Const1.de : Const1 β α → Ty.de Γ β → Ty.de Γ α
def Const2.de : Const2 γ β α → Ty.de Γ γ → Ty.de Γ β → Ty.de Γ α
def Tm.de : Tm (Ty.de Γ) α → Ty.de Γ α

def Tm.norm : (forall (Γ), Tm Γ α) → Tm Γ α
| e => quote (Tm.de (e _))

```

Lean

The `smart_bnd` function takes an additional number argument, wrapped inside a reader monad, which is used for creating fresh variables. In the paper, we leave this out and just stipulate that the variable is fresh. The same applies to `toAINF`. When discussing CSE in the paper, we describe renamings. The `rename` functions define how a renaming is applied. CSE also requires us to check equality of expressions, which is done with the `beq` functions. The CSE function in the paper also calls `lookup`, which is not defined there. It corresponds to the built-in `ListMap.lookup`. Our code also contains a function `Env.or`, which merges two environments. This is used to allow CSE to remove redundancies which appear in different, but compatible, environments.

In particular, we define the following functions in Lean, corresponding to the functions above:

```

def Prim.beq : Prim α → Prim α → Bool
def AINF.beq : AINF α → AINF α → Bool
def AINF.smart_bnd : Env → Prim α → (VPar α → Counter (AINF β)) → Counter (AINF β)
def Tm.toAINF (e : Tm VPar α) : AINF α
def Var.rename : Ren → Var α → Var α
def VPar.rename (r: Ren): VPar α → VPar α
def Env.rename (r: Ren): Env → Env
def Prim.rename (r: Ren): Prim α → Prim α
def AINF.rename (r: Ren): AINF α → AINF α
def AINF.rename (r: Ren): AINF α → AINF α

```

```

def Env.or (Γ: Env) (Δ: Env): Tern → Option Env := fun t => match Γ, Δ with
def RAINF.upgrade : RAINF → Var b → Env → Option RAINF
def AINF.cse' : Ren → RAINF → AINF α → (RAINF × VPar α)
def merge: RAINF → VPar α → AINF α

def AINF.cse : Ren → RAINF → AINF α → AINF α
| r, σ, a => let (b, c) := a.cse' r σ; merge b.reverse c

```

Lean

4.4 Proofs

In this section, we show that normalization and translation to A_i NF are type-preserving, i.e. given a well-typed term, they always produce a valid term of the same type. We also show that translation to A_i NF produces maximally fissioned terms.

We use an intrinsically typed approach where the type system of the object language is included in the encoding of the data type for the language's syntax. Therefore, the host languages type system ensures only well-typed terms can be constructed.

Following an intrinsically typed approach means that the soundness properties hold simply because our (appropriately typed) definitions type check. We do not have to state and prove explicit, separate theorems, because the types of the functions already carry the necessary information.

► **Theorem 1** (Well-typedness of Optimization).

Our optimization procedure is terminating and type preserving.

Proof. Termination is ensured by Lean's built-in termination check. The fact that normalization terminates relies on POLARA being a total language. In particular, the absence of unbounded recursion and the combination of static array sizes with intrinsic typing avoids infinite loops and out-of-bounds accesses, ensuring that our normalization function always successfully terminates. Type preservation is ensured by intrinsically-typed mechanization; consider the types of normalization and CSE in Lean:

```

def Tm.norm : (forall (Γ : Type), Tm Γ α) → Tm Γ α
| e => quote (Tm.de (e _))
def AINF.cse : Ren → RAINF → AINF α → AINF α
| r, σ, a => let (b, c) := a.cse' r σ; merge b.reverse c

```

Lean

Intrinsic typing defines the typing of the object language (here, POLARA) using the typing of the host language (here, Lean), so the host language's type checker prevents the creation of ill-typed object language programs. This means that an element of $(\forall \Gamma, \text{Tm } \Gamma \alpha)$ is a well-typed POLARA program and an element of $\text{AINF } \alpha$ is a well-typed A_i NF term. Further, given a well-typed term, each function returns a well-typed term, which is what we mean by soundness with regard to the type system. ◀

► **Theorem 2** (Well-typedness of Translation).

Our translation procedure is terminating and type preserving.

Proof. Again, termination is guaranteed by Lean's termination checker. The argument for type preservation is similar to the one above: As both POLARA and A_i NF are defined using intrinsic typing, we can only construct well-typed programs. Consider the type of `toAINF` (we omit the definition):

```
def Tm.toAINF (e : Tm VPar α) : AINF α
```

Lean

If one tried to define `toAINF` in a way that produces an ill-typed program, the definition would be rejected by the type checker. ◀

Finally, $A_i\text{NF}$ is inductively defined to be maximally fissioned, i.e., as a list of primitives without subterms, therefore the act of translating POLARA terms into $A_i\text{NF}$ in a total programming language performs loop fission by definition.

► **Theorem 3 (Maximal Fission).**

Our translation into $A_i\text{NF}$ produces terms with maximal fission.

Proof. Consider the definition of $A_i\text{NF}$ terms:

```
inductive AINF : Ty → Type
| ret : VPar α → AINF α
| bnd : Env → Var α → Prim α → AINF β → AINF β
```

Lean

Here, a value of type $VPar \alpha$ can be a variable or a parameter. A value of type $Prim \alpha$ is a primitive (not nested) operation. The constructor bnd represents a variable assignment while ret returns a variable or parameter and represents the end of the program. From this inductive definition, it is apparent that all $A_i\text{NF}$ terms have a flat structure where nested expressions are impossible. Recall that, in $A_i\text{NF}$, each assignment is considered its own separate loop. Because the body of each assignment only contains a single primitive, each loop has a body only consisting of one operation and hence an $A_i\text{NF}$ term is guaranteed to be maximally fissioned. Because the translation function toAINF has output type $AINF \alpha$, it can *only* produce such maximally fissioned terms. Further, Lean’s termination checker ensures that toAINF is total, and so always returns an $A_i\text{NF}$ term in finite time. ◀

5 Related Work

5.1 Intermediate Languages

Early work by Steele [38] implemented a continuation-passing-style (CPS) IR in a functional compiler, stressing the suitability of CPS for compilation, as it closely mimics how control flow is expressed with jumps in hardware instructions, and makes evaluation order explicit in the syntax. Appel [2] observed that beta-reductions in the lambda calculus are unsound in the presence of side effects as they could duplicate the effect. Yet, CPS, which makes evaluation order explicit, enables to perform certain optimizations, such as dead code elimination (DCE), and common subexpression elimination (CSE), by exploiting that in CPS every subterm is referenced by a unique name.

Sabry and Felleisen [34] identified that additional power of compiling in CPS [30] corresponds to the additional rules of the monadic computational language [24]. Of particular importance is the so-called associativity law of the monad, i.e., the let-let commuting conversion, enabling the flattening of code. Then, Flanagan [11] coined the name “A-normal form (ANF)” for the now popular IR, which in contrast to CPS, expresses sequential execution by simple let-binding rather than continuations. The difference between ANF and the monadic language is that ANF forbids nested let-bindings, i.e., code must be normal with regard to the associativity rule of the monad.

However, Kennedy [18] showed that moving from CPS to ANF did not take into account branching. More precisely, while the let-let commuting conversion enables the flattening of code, the let-if commuting conversion *duplicates* code into each branch, in the worst case leading to blow-up of code size exponential in the number of branches. Given that recursion always includes a branch for base case(s) and step case(s), the same problem appears with recursion. Kennedy therefore argued for a return to CPS.

The issue was resolved by Maurer et al. [22], who provided an implementation of ANF for use in the Glasgow Haskell Compiler (GHC), and further simplified by Cong et al. [6] who provided implementations for MiniScala and Lightweight-Modular-Staging (LMS). Cong showed that it is possible to combine the simplicity of let-bindings for sequential execution and the power of continuations for further control flow, by adding control operators to ANF, which enable to capture the current continuation.

In our work, we highlight the importance of commuting conversions, and extend the idea of having an intermediate language that intrinsically encodes maximal let-let conversion to an intermediate language that also intrinsically encodes maximal let-for and let-if commuting conversion, without exponential blow-up of code size. Further, as this work lies in the context of array programming, recursion is not often necessary, and can thus be avoided.

The logical connection between polarity and common subexpression elimination has also been explored by Miller and Wu [23].

5.2 Array Programming

Shaikhha et al. [37] present a differentiable programming language which is an extension of the lambda calculus. For array computations, they use an approach based on higher-order functions. It directly represents the duality of functions and arrays through built-in functions `build` for creating an array from a function and `get` for turning an array into a function. The fact that `get` is a left inverse of `build` leads to the equivalence $\text{get}(\text{build } n \ e) i \equiv e i$, which can be used for optimization. Another strand of research makes use of the standard technique of *rewriting strategies* to optimize functional array programs [13, 4], suggesting the viability of standard techniques from term rewriting for optimizing array programs. Liu et al. [20] present a framework that can express a variety of optimizations through formally verified term rewriting, achieving competitive performance; however, CSE is not addressed. Their representation is first-order and features an array generation construct similar to the one in POLARA. Optimization in POLARA is not based on rewriting, but instead uses partial evaluation.

Feldspar [3] is a DSL for array computations in Haskell. It features a `parallel` construct similar to our `for` constructor, as well as `while` loops. Feldspar is compiled to C and performs standard optimizations like fusion, as well as copy propagation and loop unrolling. Feldspar's backend uses a dataflow graph and an imperative intermediate representation, whereas our intermediate representation is functional and specifically designed to support optimizations on array programs.

SaC [35] is a functional first-order array language. Array computations are expressed using *with-loops*, which consist of at least one generator and one operator. Each generator consists of an index range and an expression giving the value of the output array at a given index in the range. The operator can provide default values for indices not included in any generator, a base array that should be modified by the generators, or it can describe an aggregation. More recently, SaC has added support for *tensor comprehensions* [36], which drop the operator part and add pattern matching on indices as well as bound and shape inference, making the notation more lightweight. Similar to our approach, their tensor comprehensions do not support summation, which is added in the form of a built-in function. SaC's optimizations are not based on a logical foundation, but consist of a pipeline of optimization algorithms. In POLARA we derive our syntax form for pattern matching on arrays from polarization type theory, enabling additional commuting conversions and thus grounding our optimization algorithm on a logical foundation.

6 Conclusion

This paper introduced $A_i\text{NF}$, a novel intermediate representation for array computations, and POLARA, a surface array language. The proposed optimization algorithm for $A_i\text{NF}$, based on typed partial evaluation and common subexpression elimination, simplifies program optimization by interpreting arrays as positively polarized types. This approach avoids complexities associated with optimization schedules for conventional ANF. We formalized $A_i\text{NF}$ and POLARA. We proved sound the translation from POLARA to $A_i\text{NF}$ and optimization.

For future work, we are working on extending the language with automatic differentiation and probabilistic primitives, and proving these extensions correct as well. We are interested in applying our optimization to redundancies generated by automatic differentiation. Further, given that $A_i\text{NF}$, based on ANF, is related to monadic notation, it would be interesting to investigate whether Applicative notation [21, 33], Arrow notation [16], and Comonad Notation [27] provide similar insights for normalization by evaluation approach to optimization.

References

- 1 Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *J. Log. Comput.*, 2(3):297–347, 1992. doi:[10.1093/LGCOM/2.3.297](https://doi.org/10.1093/LGCOM/2.3.297).
- 2 Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.
- 3 Emil Axelsson, Koen Claessen, Gergely Dévai, Zoltán Horváth, Karin Keijzer, Bo Lyckegård, Anders Persson, Mary Sheeran, Josef Svenningsson, and András Vajda. Feldspar: A domain specific language for digital signal processing algorithms. In *8th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010), Grenoble, France, 26-28 July 2010*, pages 169–178. IEEE Computer Society, 2010. doi:[10.1109/MEMCOD.2010.5558637](https://doi.org/10.1109/MEMCOD.2010.5558637).
- 4 Timon Böhler, David Richter, and Mira Mezini. Using rewrite strategies for efficient functional automatic differentiation. In Aaron Tomb, editor, *Proceedings of the 25th ACM International Workshop on Formal Techniques for Java-like Programs, FTfJP 2023, Seattle, WA, USA, 18 July 2023*, pages 51–57. ACM, 2023. doi:[10.1145/3605156.3606456](https://doi.org/10.1145/3605156.3606456).
- 5 Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 143–156. ACM, 2008. doi:[10.1145/1411204.1411226](https://doi.org/10.1145/1411204.1411226).
- 6 Youyou Cong, Leo Osvald, Grégory M. Essertel, and Tiark Rompf. Compiling with continuations, or without? whatever. *Proceedings of the ACM on Programming Languages*, 3(ICFP):79:1–79:28, 2019. doi:[10.1145/3341643](https://doi.org/10.1145/3341643).
- 7 Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021. doi:[10.1007/978-3-030-79876-5_37](https://doi.org/10.1007/978-3-030-79876-5_37).
- 8 Paul Downen and Zena M. Ariola. The duality of construction. In Zhong Shao, editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 249–269. Springer, 2014. doi:[10.1007/978-3-642-54833-8_14](https://doi.org/10.1007/978-3-642-54833-8_14).
- 9 Paul Downen and Zena M. Ariola. Compiling with classical connectives. *Log. Methods Comput. Sci.*, 16(3), 2020. URL: <https://lmcs.episciences.org/6740>.
- 10 Paul Downen and Zena M. Ariola. Duality in action (invited talk). In Naoki Kobayashi, editor, *6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina (Virtual Conference)*, volume 195 of *LIPICS*, pages 1:1–1:32. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:[10.4230/LIPICS.FSCD.2021.1](https://doi.org/10.4230/LIPICS.FSCD.2021.1).

- 11 Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In Robert Cartwright, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 237–247. ACM, 1993. doi:[10.1145/155090.155113](https://doi.org/10.1145/155090.155113).
- 12 Jeremy Gibbons. APlicative programming with Naperian functors. 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 556–583. Springer, 2017. doi:[10.1007/978-3-662-54434-1_21](https://doi.org/10.1007/978-3-662-54434-1_21).
- 13 Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. Achieving high performance the functional way: Expressing high-performance optimizations as rewrite strategies. *Commun. ACM*, 66(3):89–97, 2023. doi:[10.1145/3580371](https://doi.org/10.1145/3580371).
- 14 Charles R. Harris, K. Jarrod Millman, Stéfan van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nat.*, 585:357–362, 2020. doi:[10.1038/S41586-020-2649-2](https://doi.org/10.1038/S41586-020-2649-2).
- 15 Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 556–571. ACM, 2017. doi:[10.1145/3062341.3062354](https://doi.org/10.1145/3062341.3062354).
- 16 John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1-3):67–111, 2000. doi:[10.1016/S0167-6423\(99\)00023-4](https://doi.org/10.1016/S0167-6423(99)00023-4).
- 17 Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice Hall international series in computer science. Prentice Hall, 1993.
- 18 Andrew Kennedy. Compiling with continuations, continued. In Ralf Hinze and Norman Ramsey, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pages 177–190. ACM, 2007. doi:[10.1145/1291151.1291179](https://doi.org/10.1145/1291151.1291179).
- 19 Neelakantan R. Krishnaswami. Focusing on pattern matching. 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 366–378. ACM, 2009. doi:[10.1145/1480881.1480927](https://doi.org/10.1145/1480881.1480927).
- 20 Amanda Liu, Gilbert Louis Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. Verified tensor-program optimization via high-level scheduling rewrites. *Proc. ACM Program. Lang.*, 6(POPL):1–28, 2022. doi:[10.1145/3498717](https://doi.org/10.1145/3498717).
- 21 Simon Marlow, Simon Peyton Jones, Edward Kmett, and Andrey Mokhov. Desugaring Haskell’s do-notation into applicative operations. In Geoffrey Mainland, editor, *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*, pages 92–104. ACM, 2016. doi:[10.1145/2976002.2976007](https://doi.org/10.1145/2976002.2976007).
- 22 Luke Maurer, Paul Downen, Zena M. Ariola, and Simon L. Peyton Jones. Compiling without continuations. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 482–494. ACM, 2017. doi:[10.1145/3062341.3062380](https://doi.org/10.1145/3062341.3062380).
- 23 Dale Miller and Jui-Hsuan Wu. A positive perspective on term representation (invited talk). In Bartek Klin and Elaine Pimentel, editors, *31st EACSL Annual Conference on Computer Science Logic, CSL 2023, February 13-16, 2023, Warsaw, Poland*, volume 252 of *LIPICS*, pages 3:1–3:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:[10.4230/LIPICS.CSL.2023.3](https://doi.org/10.4230/LIPICS.CSL.2023.3).

- 24 Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, pages 14–23. IEEE Computer Society, 1989. doi:10.1109/LICS.1989.39155.
- 25 Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Trans. Comput. Log.*, 9(3):23:1–23:49, 2008. doi:10.1145/1352582.1352591.
- 26 nLab authors. function type. <https://ncatlab.org/nlab/show/function+type>, January 2024. Revision 33.
- 27 Dominic A. Orchard and Alan Mycroft. A notation for comonads. In Ralf Hinze, editor, *Implementation and Application of Functional Languages - 24th International Symposium, IFL 2012, Oxford, UK, August 30 - September 1, 2012, Revised Selected Papers*, volume 8241 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2012. doi:10.1007/978-3-642-41582-1_1.
- 28 Adam Paszke, Daniel D. Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew J. Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. Getting to the point: index sets and parallelism-preserving autodiff for pointful array programming. *Proc. ACM Program. Lang.*, 5(ICFP):1–29, 2021. doi:10.1145/3473593.
- 29 Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In Richard L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22–24, 1988*, pages 199–208. ACM, 1988. doi:10.1145/53990.54010.
- 30 Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975. doi:10.1016/0304-3975(75)90017-1.
- 31 Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman P. Amarasinghe, and Frédéric Durand. Halide: decoupling algorithms from schedules for high-performance image processing. *Commun. ACM*, 61(1):106–115, 2018. doi:10.1145/3150211.
- 32 David Richter, Timon Böhler, Pascal Weisenburger, and Mira Mezini. stg-tud/ainf-compiling-with-arrays. Software, swID: [swh:1:dir:8e0e755d11e4e3e91fb05bf8df1a5c8bec0f553a](https://github.com/stg-tud/ainf-compiling-with-arrays) (visited on 2024-09-02). URL: <https://github.com/stg-tud/ainf-compiling-with-arrays>.
- 33 David Richter, Timon Böhler, Pascal Weisenburger, and Mira Mezini. A direct-style effect notation for sequential and parallel programs. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17–21, 2023, Seattle, Washington, United States*, volume 263 of *LIPICS*, pages 25:1–25:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.ECOOP.2023.25.
- 34 Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *LISP Symb. Comput.*, 6(3-4):289–360, 1993.
- 35 Sven-Bodo Scholz. Single Assignment C: efficient support for high-level array operations in a functional setting. *J. Funct. Program.*, 13(6):1005–1059, 2003. doi:10.1017/S0956796802004458.
- 36 Sven-Bodo Scholz and Artjoms Sinkarovs. Tensor comprehensions in SaC. In Jurriën Stutterheim and Wei-Ngan Chin, editors, *IFL '19: Implementation and Application of Functional Languages, Singapore, September 25–27, 2019*, pages 15:1–15:13. ACM, 2019. doi:10.1145/3412932.3412947.
- 37 Amir Shaikhha, Andrew W. Fitzgibbon, Dimitrios Vytiniotis, and Simon Peyton Jones. Efficient differentiable programming in a functional array-processing language. *Proc. ACM Program. Lang.*, 3(ICFP):97:1–97:30, 2019. doi:10.1145/3341701.
- 38 Guy L. Steele. Rabbit: A compiler for Scheme. Technical report, Massachusetts Institute of Technology, USA, 1978.
- 39 Noam Zeilberger. On the unity of duality. *Ann. Pure Appl. Log.*, 153(1-3):66–96, 2008. doi:10.1016/J.APAL.2008.01.001.
- 40 Noam Zeilberger. *The logical basis of evaluation order and pattern-matching*. PhD thesis, Carnegie Mellon University, USA, 2009. AAI3358066.

Pipit on the Post: Proving Pre- and Post-Conditions of Reactive Systems

Amos Robinson  

Sydney, Australia

Alex Potanin  

Australian National University, Canberra, Australia

Abstract

Synchronous languages such as Lustre and Scade are used to implement safety-critical control systems; proving such programs correct and having the proved properties apply to the compiled code is therefore equally critical. We introduce Pipit, a small synchronous language embedded in F^{*}, designed for verifying control systems and executing them in real-time. Pipit includes a verified translation to transition systems; by reusing F^{*}'s existing proof automation, certain safety properties can be automatically proved by k-induction on the transition system. Pipit can also generate executable code in a subset of F^{*} which is suitable for compilation and real-time execution on embedded devices. The executable code is deterministic and total and preserves the semantics of the original program.

2012 ACM Subject Classification Computer systems organization → Real-time languages; Theory of computation → Program verification; Software and its engineering → Specialized application languages

Keywords and phrases Lustre, streaming, reactive, verification

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.34

Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact):*

<https://doi.org/10.4230/DARTS.10.2.19>

Software (source code and mechanised proofs): <https://github.com/songlarknet/pipit>

archived at sw.h1:dir:8839600ca8830ab20681ed03760f642bf877b77e

1 Introduction

Safety-critical control systems, such as the anti-lock braking systems that are present in most cars today, need to be correct and execute in real-time. One approach, favoured by parts of the aerospace industry, is to implement the controllers in a high-level language such as Lustre [10] or Scade [13], and verify that the implementations satisfy the high-level specification using a model-checker, such as Kind2 [11]. These model-checkers can prove many interesting safety properties automatically, but do not provide many options for manual proofs when the automated proof techniques fail. Additionally, the semantics used by the model-checker may not match the semantics of the compiled code, in which case properties proved do not necessarily hold on the real system. This mismatch may occur even when the compiler has been verified to be correct, as in the case of Vélus [5]. For example, in Vélus, integer division rounds towards zero, matching the semantics of C; however, integer division in Kind2 rounds to negative infinity, matching SMT-lib [2, 25].

To be confident that our proofs hold on the real system, we need a single shared semantics for the compiler and the prover. In this paper we introduce Pipit¹, an embedded domain-specific language for implementing and verifying controllers in F^{*}. Pipit aims to provide a

¹ Implementation available at <https://github.com/songlarknet/pipit>

 © Amos Robinson and Alex Potanin;
licensed under Creative Commons License CC-BY 4.0
38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 34; pp. 34:1–34:28



high-level language based on Lustre, while reusing F^{*}'s proof automation and manual proofs for verifying controllers [31], and using Low^{*}'s C-code generation for real-time execution [34]. To verify programs, Pipit translates its expression language to a transition system for k-inductive proofs, which is verified to be an abstraction of the original semantics. To execute programs, Pipit can generate executable code, which is total and semantics-preserving.

In this paper, we make the following contributions:

- we motivate the need to combine manual and automated proofs of reactive systems with a strong specification language (Section 2);
- we introduce Pipit, a minimal synchronous language that supports rely-guarantee contracts and properties; crucially, proof obligations are annotated with a status – *valid* or *deferred* – allowing proofs to be delayed until more is known of the program context (Section 3);
- we describe a *checked semantics* for Pipit; after checking deferred properties, programs are *blessed*, which marks their properties as valid (Subsection 3.2);
- we describe an encoding of transition systems that can express under-specified rely-guarantee contracts as functions rather than relations; composing functions results in simpler transition systems (Section 4);
- we identify the invariants and lemmas required to prove that the abstract transition system is an abstraction of the original semantics (Subsection 3.3, Subsection 4.3);
- similarly, we offer a mechanised proof that the executable transition system preserves the original semantics (Section 5);
- finally, we evaluate Pipit by implementing the high-level logic of a Time-Triggered Controller Area Network (TTCAN) bus driver and verifying an abstract model of a key component (Section 6).

2 Pipit for time-triggered networks

To introduce Pipit, we consider a *time-triggered* network driver, which has a static schedule dictating the network traffic, and which all nodes on the network must adhere to. This driver is a simplification of the Time-Triggered Controller Area Network (TTCAN) bus specification [15] which we will discuss further in Section 6.

At a high level, the network schedule is described by a *system matrix* which consists of rows of *basic cycles*. Each basic cycle consists of a sequence of actions to be performed at specific time-marks. Actions in the schedule may not be relevant to all nodes; the node's *node matrix* contains only the relevant actions. The node matrix is represented in memory by a *triggers array* containing triggers sorted by their time-marks; trigger actions include sending and receiving application-specific messages, sending reference messages, and triggering “watch” alerts. Reference messages start a new basic cycle; a subset of nodes, designated as leaders, send reference messages to synchronise the network. Watch alerts are generally placed after an expected reference message to signal an error if no reference message is received.

Figure 1 (left) shows an example node matrix for a non-leader node. The matrix consists of two basic cycles C0 and C1 with messages sent at time-marks 0, 1 and 2. The node expects to receive a reference message at time-mark 7; the watch at time-mark 9 allows a grace period before triggering an error if the reference message is not received. Figure 1 (right) shows the corresponding triggers array.

The network has strict timing requirements which prohibit the driver from looping through the entire triggers array at each time-mark. Instead, the driver maintains an index that refers to the current trigger. At each time-mark, the driver checks if the current trigger has expired or is inactive, and if so, it increments the index.

	TM0	TM1	TM2	...	TM9	0:{ time = 0; enabled = {C0,C1}; action = SEND(A); }
C0	SEND A	SEND B	-	...	WATCH	1:{ time = 1; enabled = {C0}; action = SEND(B); }
C1	SEND A	-	SEND C	...	WATCH	2:{ time = 2; enabled = {C1}; action = SEND(C); }

Figure 1 Left: node matrix; right: corresponding triggers array configuration.

2.1 Deferring and proving properties

We implement a streaming function *count_when* to maintain the index into the triggers array; the function takes a constant natural number *max* and a stream of booleans *inc*. At each step, *count_when* checks whether the current increment flag is true; if so, it increments the previous counter, saturating at the maximum; otherwise, it leaves the counter as-is.

```
let count_when (max: N) (inc: stream B): stream N =
  rec count
    check? (0 ≤ count ≤ max);
    let count' = (0 fby count) + (if inc then 1 else 0) in
      if count' ≥ max then max else count'
```

The implementation of *count_when* first defines a recursive stream, *count*, which states an invariant about the count before defining the incremented stream *count'*. Inside *count'*, the syntax *0 fby count* is read as “the initial value of zero followed by the previous count”.

The syntax *check_? (0 ≤ count ≤ max)* asserts that the count is within the range [0, *max*]. The subscript _? on the check is the *property status*, which in this case denotes that the assertion has been stated, but it is not yet known whether it holds. A property status of _?, on the other hand, denotes that a property has been proved to hold. These property statuses are used to defer checking properties until enough is known about the environment, and to avoid rechecking properties that have already been proven. In practice, the user does not explicitly specify property statuses in the source language. The stated property (*0 ≤ count ≤ max*) is a stream of booleans which must always be true. Non-streaming operations such as *≤* are implicitly lifted to streaming operations, and non-streaming values such as 0 and *max* are implicitly lifted to constant streams.

We defer the proof of the property here because, at the point of stating the property inside the *rec* combinator, we don’t yet have a concrete definition for the *count* variable. In this case, we could have instead deferred the *statement* of the property by introducing a let-binding for the recursive count and putting the *check* outside of the *rec* combinator. However, it is not always possible to defer property statements: for example, when calling other streaming functions that have their own preconditions, it may not be possible to move the function call outside of its enclosing *rec*.

Pipit is an embedded domain-specific language. The program above is really syntactic sugar for an F^{*} program that takes a natural number and constructs a Pipit core expression with a free boolean variable. We will discuss the details of the core language in Section 3, but for now we focus on the source program with some minor embedding details omitted.

To actually prove the property above, we use the meta-language F^{*}’s tactics to translate the program into a transition system and prove the property inductively on the system. Finally, we *bless* the expression, which marks the properties as valid ([_? := _?]). Blessing is an intensional operation that traverses the expression and updates the internal metadata, but does not affect the runtime semantics.

```
let count_when? (max: N): stream B → stream N =
  let system = System.translate1(count_when max) in
  assert (System.inductive_check system) by (pipit_simplify ());
  bless1 (count_when max)
```

The subscript 1 in the translation to transition system and blessing operations refers to the fact that the stream function has one stream parameter. The *pipit_simplify* tactic in the assertion performs normalisation-by-evaluation to simplify away the translation to a first-order transition system; F*'s proof-by-SMT can then solve the inductive check directly.

Callers of *count_when* can now use the validated variant without needing to re-prove the count-range property. In a dedicated model-checker such as Kind2 [11] or Lesar [35], this kind of bookkeeping would all be performed under-the-hood. By embedding Pipit in a general-purpose theorem prover, we move some of the bookkeeping burden onto the user; however, we have increased confidence that the compiled code matches the verified code and, as we shall see, we also have access to a rich specification language.

2.2 Restrictions on the triggers array

Our driver may fall behind when trying to execute certain schedules, as the driver only processes one trigger per time-mark. To ensure that the schedule can be executed on time, the triggers array must allow sufficient time for the driver to skip over any disabled triggers before the next enabled trigger starts.

Recall our concrete triggers array from Figure 1, which contained trigger 1 (SEND B at time-mark 1 on cycle C0), and trigger 2 (SEND C at time-mark 2 on cycle C1). We could postpone trigger 1 to send B at time-mark 2, as the corresponding cell in the node matrix is empty. However, we *cannot* bring the trigger at index 2 forward to send message C at time-mark 1, as it takes two steps to reach trigger 2 from the start of the array.

We impose three restrictions on *valid* triggers arrays: the time-marks must be sorted; there must be an adequate time-gap between any two triggers that are enabled on the same cycle index; and each trigger's time-mark must be greater-than-or-equal to its index, so that it is reachable in time from the start of the array.

With these restrictions in place, we prove a lemma *lemma_can_reach_next*, which states that for all valid cycle indices and trigger indices, if the current trigger is enabled in the current cycle and there is another enabled trigger scheduled to occur somewhere in the array after the current one, then there is an adequate time-gap to allow the driver to skip over any disabled triggers in-between. These properties are straightforward in a theorem prover, but are difficult to state in a model-checker with a limited specification language.

2.3 Instantiating lemmas and defining contracts

We can now implement the trigger-fetch logic, which keeps track of the current trigger. We use the *count_when* streaming function to define the index of the current trigger; we tell *count_when* to increment the index whenever the previous index has expired or is inactive in the current basic cycle. We simplify our presentation here and only consider a constant cycle: the real system presented in Section 6 has some extra complexity such as resetting the index, incrementing the cycle index at the start of a new cycle, and using machine integers.

```
let trigger_fetch (cycle: N) (time: stream N): stream N =
  rec index.
    let inc = false fby ((time_mark index) ≤ time ∨ ¬(enabled index cycle)) in
    let index = count_when[trigger_count inc] in
    pose (lemma_can_reach_next cycle index);
    check[ ] (can_reach_next_active cycle time index);
    index
```

The *trigger_fetch* function takes a static cycle index and a stream denoting the current time. The increment flag and the index are mutually dependent – the increment flag depends on the previous value of the index, while the index depends on the current value of the increment flag – so we introduce a recursive stream for the index. We allow the index to go one past the end of the array to denote that there are no more triggers.

We use the *pose* helper function to lift the *lemma_can_reach_next* lemma to a streaming context and instantiate it. We then state an invariant as a deferred property. Informally, the invariant states that, either the current active trigger is not late, or the next active trigger after the current index is in the future and we can reach it in time.

With the explicitly instantiated lemma, we can prove the streaming invariant by straightforward induction on the transition system. To help compose this function with the rest of the system, we also abstract over the details of the trigger-fetch mechanism by introducing a rely-guarantee contract for *trigger_fetch*. The contract we state is that if we are called once per time-mark then we guarantee that we never encounter a late trigger.

```
let trigger_fetch□ (cycle: N): stream N → stream N =
  let contract = Contract.contract_of_stream1 {
    rely = (λtime. time = 0 fby (time + 1));
    guar = (λtime index. (index_valid index ∧ enabled index cycle)
              ⇒ (time_mark index) ≥ time);
    body = (λtime. trigger_fetch cycle time);
  } in
  assert (Contract.inductive_check contract) by (pipit_simplify ());
  Contract.stream_of_contract1 contract
```

In the implementation of the validated variant of *trigger_fetch*, we first construct the contract from streaming functions. The *Contract.contract_of_stream₁* combinator describes a contract with one input (the time stream), and takes stream transformers for each of the rely, guarantee and body. The combinator transforms the surface syntax into core expressions. The assertion (*Contract.inductive_check contract*) then translates the expressions into a transition system, and checks that if the rely always holds then the guarantee always holds, and that the as-yet-unchecked subproperties hold. Finally, *Contract.stream_of_contract₁* blesses the core expression and converts it back to a stream transformer, so it can be easily used by other parts of the program.

The key distinction between our streaming rely-guarantee contracts and imperative pre-post contracts is that the rely and guarantee are both *streams* of booleans, rather than instantaneous predicates. In this case, the rely (*time = 0 fby (time + 1)*) checks that the current time is exactly one time-mark after the time at the previous *tick* of computation. Expressing such a rely in an imperative setting would require extra encoding, as preconditions in imperative languages do not generally have an innate notion of the previous value with respect to a global shared clock.

When *trigger_fetch* is used in other parts of the program, the caller must ensure that the environment satisfies the rely clause. In the core language, this is tracked by another deferred property status attached to the contract; we will discuss this further in Section 3.

3 Pipit core language

We now introduce the core Pipit language. Note that this form differs slightly from the surface syntax presented earlier in Section 2, which used the syntax of the metalanguage F*, as well as including proofs in F* itself.

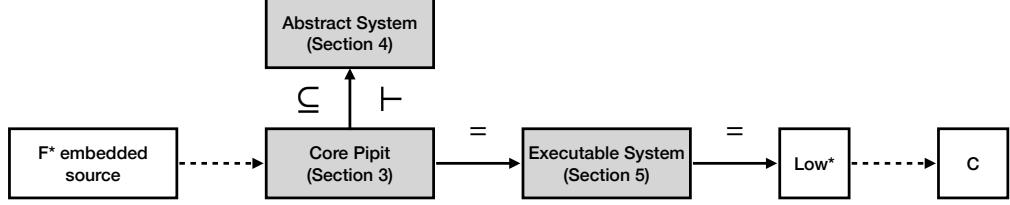


Figure 2 Architecture of Pipit. The gray boxes and solid arrows are defined in this paper. The white boxes and dashed arrows are trusted components. The labels denote verified properties of the translation: abstraction (\subseteq), entailment of proof obligations (\vdash), and equivalence (=).

$e, e' := v \mid x \mid p(\bar{e})$	(values, variables and operations)
$v \text{ fby } e \mid \text{rec } x. e[x]$	(delayed and recursive streams)
$\text{let } x = e \text{ in } e'[x]$	(let-expressions)
$\text{check}_\pi e_{\text{prop}}$	(checked properties)
$\text{contract}_\pi \{e_{\text{rely}}\} e_{\text{body}} \{x. e_{\text{guar}}[x]\}$	(rely-guarantee contracts)
$v := n \in \mathbb{N} \mid b \in \mathbb{B} \mid r \in \mathbb{R} \mid \dots$	(values)
$p := (+) \mid (-) \mid (\times) \mid \text{if-then-else} \mid \dots$	(primitives)
$\pi := \checkmark \mid \text{?}$	(property statuses: valid or unknown)
$V := \cdot \mid V; v$	(streams of values)
$\sigma := \{x \mapsto v\}$	(heaps)
$\Sigma := \cdot \mid \Sigma; \sigma$	(streaming history environments)
$\tau, \tau' := \mathbb{N} \mid \mathbb{B} \mid \tau \times \tau \mid \dots$	(value types)
$\Gamma := \cdot \mid x : \tau, \Gamma$	(type environments)

Figure 3 Core grammar: expressions e , values v , primitive operations p , and property statuses π .

Figure 2 shows the high-level architecture of Pipit. On the left-hand-side, the surface syntax embedded in F^* is shown; this includes some Pipit-specific syntactic sugar. The translation from the surface syntax to the core language is trusted. There are two targets from the core language: abstract transition systems for verification, and executable transition systems for extraction to C. The translation to abstract systems is verified to be an abstraction according to the dynamic semantics (Subsection 3.1). The translation to abstract systems also generates proof obligations, which are verified to correspond to the proof obligations on the original program. The translation to executable transition systems is proven to be semantics-preserving, as is the subsequent translation to Low^* . The translation from Low^* to C is external to this paper and forms part of our trusted computing base.

Figure 3 defines the grammar of Pipit. The expression form e includes standard syntax for values (v), variables (x) and primitive applications ($p(\bar{e})$). Most of the expression forms were introduced informally in Section 2 and correspond to the clock-free expressions of Lustre [10].

The expression syntax for delayed streams ($v \text{ fby } e$) denotes the previous value of the stream e , with an initial value of v when there is no previous value.

Recursive streams are defined using the fixpoint operator (`rec` x . $e[x]$); the syntax $e[x]$ means that the variable x can occur in e . As in Lustre, recursive streams can only refer to their previous values and must be *guarded* by a delay: the stream (`rec` x . 0 `fby` $(x + 1)$) is well-defined and counts from zero up, but the stream (`rec` x . $x + 1$) is invalid and has no computational interpretation. This form of recursion differs slightly from standard Lustre, which uses a set of mutually-recursive bindings. Although we cannot express mutually-recursive bindings in the core syntax here, we can express them as a notation on the surface syntax by combining the bindings together into a record or tuple.

Checked properties and contracts are annotated with their property status π , which can either be valid (✓) or unknown (✗). For checked properties `check π e`, the property status denotes whether the property has been proved to be valid.

Contracts `contract π { e_{rely} } e_{body} { x . $e_{\text{guar}}[x]$ }` allow modular reasoning by replacing the implementation with an abstract specification. Contracts involve two verification conditions. Firstly, when a contract is *defined*, the definer must prove that the body satisfies the contract: roughly, if e_{rely} is always true, then $e_{\text{guar}}[x := e_{\text{body}}]$ is always true. Secondly, when a contract is *instantiated*, the caller must prove that the environment satisfies the precondition: that is, e_{rely} is always true. Conceptually, then, a contract could have two property statuses: one for the definition and one for the instantiation. However, in practice, it is not useful to defer the proof of a contract definition – one could achieve a similar effect by replacing the contract with its implementation. For this reason, we only annotate contracts with one property status, which denotes whether the instantiation has been proved to satisfy the precondition.

For example, the core expression (`rec sum. (0 fby sum) + ints`) computes the sum of values from a stream of integers ints by defining a recursive stream sum , which is delayed and given an initial value of zero. If we were to use this sum in a context that required a strictly positive integer, we could give it a contract that states that if the input stream is always positive, then the resulting sum is also positive:

```
contract✗ {ints > 0} (rec sum. (0 fby sum) + ints) {sum. sum > 0}
```

To be considered a valid program, we must prove that the contract definition itself holds, as with our earlier contract (Subsection 2.3). The unknown property status here allows us to defer the caller’s proof that the input stream is always positive until the contract is used.

The remaining grammatical constructs of Figure 3 describe streams, value environments, types and type environments. Streams V are represented as a sequence of values; streaming history environments Σ are streams of heaps. Types τ and type environments Γ are standard. For the presentation of the formal grammar here, we consider only a fixed set of values and primitives; in practice, the implementation is parameterised by a primitive table which we extend with immutable array operations for the TTCAN driver logic in Section 6.

We define the typing judgments for Pipit in Figure 4. Most of the typing rules are standard for an unclocked Lustre. The typing judgment $\Gamma \vdash e : \tau$ denotes that, in an environment of streams Γ , expression e denotes a stream of type τ . This core typing judgment differs from the surface syntax used in Section 2, which used an explicit stream type; for the core language, we instead assume that everything is a stream.

We use an auxiliary function $\text{prim-value-type}(v) = \tau$ to denote that value v has type τ ; for primitives $\text{prim-type}(p) = (\tau_1 \times \dots \times \tau_n) \rightarrow \tau'$ denotes that p takes arguments of type τ_i and returns a result of type τ' . Primitives are pure, non-streaming functions.

Rules TVALUE, TVAR, TPRIM and TLET are standard.

Rule TFBY states that expression $v \text{ fby } e$ requires both v and e to have equal types.

Rule TREC states that a recursive stream `rec` x . e has the recursive stream bound inside e . The recursion must also be guarded, in that any recursive references to x are delayed, but this requirement is performed as a separate syntactic check described in Subsection 3.3.

$$\boxed{\Gamma \vdash e : \tau}$$

$$\begin{array}{c}
\frac{\text{prim-value-type}(v) = \tau}{\Gamma \vdash v : \tau} \text{ (TVALUE)} \quad \frac{}{\Gamma, x : \tau, \Gamma' \vdash x : \tau} \text{ (TVAR)} \\
\\
\frac{\text{prim-type}(p) = (\tau_1 \times \dots \times \tau_n) \rightarrow \tau' \quad \Gamma \vdash e_1 : \tau_1 \quad \dots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash p(\bar{e}) : \tau'} \text{ (TPRIM)} \\
\\
\frac{\text{prim-value-type}(v) = \tau \quad \Gamma \vdash e' : \tau}{\Gamma \vdash v \text{ fby } e' : \tau} \text{ (TFBY)} \quad \frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \text{rec } x. e[x] : \tau} \text{ (TREC)} \\
\\
\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash e' : \tau'}{\Gamma \vdash \text{let } x = e \text{ in } e'[x] : \tau'} \text{ (TLET)} \quad \frac{\Gamma \vdash e : \mathbb{B}}{\Gamma \vdash \text{check}_\pi e : \text{unit}} \text{ (TCHECK)} \\
\\
\frac{\Gamma \vdash e_{\text{rely}} : \mathbb{B} \quad \Gamma \vdash e_{\text{body}} : \tau \quad \Gamma, x : \tau \vdash e_{\text{guar}} : \mathbb{B}}{\Gamma \vdash \text{contract}_\pi \{e_{\text{rely}}\} e_{\text{body}} \{x. e_{\text{guar}}[x]\} : \tau} \text{ (TCONTRACT)}
\end{array}$$

Figure 4 Typing rules for Pipit; the judgment $\Gamma \vdash e : \tau$ denotes that expression e describes a stream of values of type τ . Auxiliary functions are used for values and primitive operations.

Rule TCHECK states that checked properties $\text{check}_\pi e$ require a boolean property e .

Finally, rule TCONTRACT applies for a contract $\text{contract}_\pi \{e_{\text{rely}}\} e_{\text{body}} \{x. e_{\text{guar}}[x]\}$ with a body expression of type τ . The overall expression has result type τ . Both rely and guarantee must be boolean expressions, and the guarantee can refer to the result as x .

3.1 Dynamic semantics

The dynamic semantics of Pipit are defined in Figure 5. We present our semantics in a big-step form. This differs somewhat from traditional *reactive* semantics of Lustre [10]. Our big-step semantics emphasises the equational nature of Pipit, as it is substitution-based and syntax-directed, while the reactive semantics emphasises the finite-state streaming execution of the system. We use transition systems for reasoning about the finite-state execution (Section 4), which is fairly standard [9, 11, 35]. Previous work on the W-CALCULUS [17] for linear digital-signal-processing filters makes a similar distinction and provides a non-streaming semantics for reasoning about programs and a streaming semantics for executing programs.

The judgment form $\Sigma \vdash e \Downarrow v$ denotes that expression e evaluates to value v under streaming history Σ . The streaming history is a stream of heaps; in practice, we only evaluate expressions with a non-empty streaming history.

At a high level, evaluation unfolds recursive streams to determine a value. For example, to evaluate the earlier sum example with input $\text{ints} = [1; 2]$, we start with the judgment:

$$\{ \text{ints} \mapsto 1 \}; \{ \text{ints} \mapsto 2 \} \vdash (\text{rec } \text{sum}. (0 \text{ fby } \text{sum}) + \text{ints}) \Downarrow v$$

First, we unfold the recursive stream one step to get $(0 \text{ fby } (\text{rec } \text{sum}. (0 \text{ fby } \text{sum}) + \text{ints})) + \text{ints}$. Evaluation of primitives is standard. To evaluate variables, we look for the variable in the current (rightmost) heap:

$$\frac{}{\{ \text{ints} \mapsto 1 \}; \{ \text{ints} \mapsto 2 \} \vdash \text{ints} \Downarrow 2} \text{ (VAR)}$$

$$\begin{array}{c}
\boxed{\Sigma \vdash e \Downarrow v} \\
\begin{array}{ccc}
\frac{}{\Sigma; \sigma \vdash x \Downarrow \sigma(x)} \text{ (VAR)} & \frac{}{\Sigma \vdash v \Downarrow v} \text{ (VALUE)} & \frac{\Sigma \vdash e'[x := e] \Downarrow v}{\Sigma \vdash \text{let } x = e \text{ in } e'[x] \Downarrow v} \text{ (LET)} \\
\\
\frac{\Sigma \vdash e_1 \Downarrow v_1 \quad \dots \quad \Sigma \vdash e_n \Downarrow v_n}{\Sigma \vdash p(\bar{e}) \Downarrow \text{prim-sem}(p, \bar{v})} \text{ (PRIM)} & & \\
\\
\frac{}{\sigma \vdash v \text{ fby } e' \Downarrow v} \text{ (FBY}_1\text{)} & \frac{\text{length}(\Sigma) > 0 \quad \Sigma \vdash e' \Downarrow v'}{\Sigma; \sigma \vdash v \text{ fby } e' \Downarrow v'} \text{ (FBY}_S\text{)} & \\
\\
\frac{\Sigma \vdash e[x := \text{rec } x. e] \Downarrow v}{\Sigma \vdash \text{rec } x. e[x] \Downarrow v} \text{ (REC)} & & \frac{}{\Sigma \vdash \text{check}_{\pi} e \Downarrow ()} \text{ (CHECK)} \\
\\
\frac{\Sigma \vdash e_{\text{body}} \Downarrow v}{\Sigma \vdash \text{contract}_{\pi} \{e_{\text{rely}}\} e_{\text{body}} \{x. e_{\text{guar}}[x]\} \Downarrow v} \text{ (CONTRACT)} & & \\
\\
\boxed{\Sigma \vdash e \Downarrow^* V} & & \boxed{\Sigma \vdash e \Downarrow^{\square} \top} \\
\\
\frac{}{\cdot \vdash e \Downarrow^*.} \text{ (STEPS}_0\text{)} & \frac{\Sigma \vdash e \Downarrow V \quad \Sigma; \sigma \vdash e \Downarrow v}{\Sigma; \sigma \vdash e \Downarrow V; v} \text{ (STEPS}_S\text{)} & \\
\\
\frac{\Sigma \vdash e \Downarrow^* \top; \dots}{\Sigma \vdash e \Downarrow^{\square} \top} \text{ (ALWAYS)} & &
\end{array}
\end{array}$$

Figure 5 Dynamic semantics for Pipit; the judgment form $\Sigma \vdash e \Downarrow v$ denotes that evaluating expression e under streaming history Σ results in value v .

For delays, we discard the current heap and continue evaluation with the history prefix:

$$\frac{\{ints \mapsto 1\} \vdash (\text{rec } sum. (0 \text{ fby } sum) + ints) \Downarrow 1 \quad \{ints \mapsto 1\}; \{ints \mapsto 2\} \vdash 0 \text{ fby } (\text{rec } sum. (0 \text{ fby } sum) + ints) \Downarrow 1}{\{ints \mapsto 1\}; \{ints \mapsto 2\} \vdash 0 \text{ fby } (\text{rec } sum. (0 \text{ fby } sum) + ints) \Downarrow 1} \text{ (FBY}_S\text{)}$$

Returning to Figure 5, rule VAR evaluates a variable x under some non-empty stream history $\Sigma; \sigma$, where σ is the most recent heap. Rules VALUE and LET are standard. Rule PRIM evaluates a primitive p applied to many arguments e_1 to e_n by evaluating each argument separately; we then apply the primitive with prim-sem metafunction.

For delay expressions $v \text{ fby } e$, we have two cases depending on whether there is a previous value. When there is no previous value – the streaming history only contains the current heap – rule FBY₁ evaluates to the default value v . Otherwise, rule FBY_S applies; we evaluate the previous value of e by discarding the most recent entry from the streaming history.

Rule REC evaluates a recursive stream $\text{rec } x. e$ by unfolding the recursion one step. For causal expressions (Subsection 3.3), where each recursive occurrence of x is guarded by a followed-by, this unfolding eventually terminates as each followed-by shortens the history.

Rule CHECK ignores the property when evaluating check expressions. We do not dynamically check the property here; this is done in the checked semantics (Subsection 3.2).

Similarly, rule CONTRACT ignores preconditions and postconditions when evaluating contracts. From an abstraction perspective, it would be valid to return an arbitrary value that satisfies the contract. However, such an abstraction would make evaluation non-deterministic and, for contracts with unsatisfiable postconditions, non-total. The deterministic and total nature of evaluation is key to our proofs and metatheory.

We also define two auxiliary judgment forms: $\Sigma \vdash e \Downarrow^* V$ and $\Sigma \vdash e \Downarrow^\square \top$.

Judgment form $\Sigma \vdash e \Downarrow^* V$ denotes that, under history Σ , expression e evaluates to the stream V . This judgment performs iterated application of single-value evaluation.

Judgment form $\Sigma \vdash e \Downarrow^\square \top$ denotes that a boolean expression e evaluates to the stream of trues under history Σ . Informally, it can be read as “ e is always true in history Σ ”.

3.2 Checked semantics

In addition to the big-step semantics above, we also define a judgment form for checking that the properties and contracts of a program hold for a particular streaming history. We call these the *checked* semantics; they are comparable to checking runtime assertions.

The checked semantics have the judgment form $\Sigma \vdash_\pi e$ valid, which denotes that under streaming history Σ , the properties and contracts of e with status π hold. The property status dictates which properties should be checked and which should be ignored.

We consider a program to be *valid* if its checks hold for all histories ($\forall \Sigma. \Sigma \vdash_\square e$ valid). The checked semantics are a specification describing what it means to be a valid program. We do not generally verify programs directly using the checked semantics; instead, we translate to an abstract transition system and construct the proofs there (Section 4).

To check a property ($\text{check}_\pi e$) in history Σ , we check that e is always true ($\Sigma \vdash e \Downarrow^\square \top$).

Checking contracts is more involved. For whole-program correctness, it would suffice to check that a contract’s rely and guarantee both hold. However, the purpose of contracts is to enable modular reasoning about parts of the program: we need to be able to check contracts independently of their context. Conceptually, then, contracts involve two kinds of checks: one for the definition and one for the call-site. To check a contract definition, we check that the body satisfies the guarantee for all *valid* contexts – that is, those where the rely holds. Then, to check a contract instance, we just need to check that the call-site satisfies the rely.

For example, recall our earlier contract that the sum of strictly positive integers is positive:

```
let sum i = contract[ ] {i > 0} (rec sum. (0 fby sum) + i) {sum. sum > 0}
```

To check the contract definition on a concrete input $i = [1; 2]$, we first evaluate the body:

$$\{i \mapsto 1\}; \{i \mapsto 2\} \vdash (\text{rec } sum. (0 \text{ fby } sum) + i) \Downarrow^* [1; 3]$$

We then check that, assuming all inputs are positive, then all results are positive:

$$\{i \mapsto 1\}; \{i \mapsto 2\} \vdash i > 0 \Downarrow^\square \top \implies \{i \mapsto 1, sum \mapsto 1\}; \{i \mapsto 2, sum \mapsto 3\} \vdash sum > 0 \Downarrow^\square \top$$

It is critical that the rely is true *at all points* in the stream. Consider if we had instead used the input stream $i = [-10; 1]$; the rely is false at the first step, but is instantaneously true at the second step. In this case, the sum is -10 at the first step, and -9 at the second step. At both steps the output is negative and the guarantee is false, even though the rely becomes true at the second step. The contract itself remains valid, however, as the assumption is invalid: the input did not satisfy the rely at all steps.

The checked semantics of Pipit is defined in Figure 6.

Rules CHKVALUE and CHKVAR state that values and variables are always valid.

Rule CHKPRIM checks a primitive application by descending into the subexpressions. Similarly, rule CHKFBY descends into followed-by expressions.

Rule CHKREC checks a recursive-expression `rec` $x. e$ by evaluating the overall expression to a stream of values V . The rule then extends the streaming environment Σ with x bound to the values from V ; this extended environment is used to descend into the recursive expression.

$$\begin{array}{c}
 \boxed{\Sigma \vdash_{\pi} e \text{ valid}} \\
 \frac{}{\Sigma \vdash_{\pi} v \text{ valid}} \text{ (CHKVALUE)} \quad \frac{}{\Sigma \vdash_{\pi} x \text{ valid}} \text{ (CHKVAR)} \\
 \frac{\Sigma \vdash_{\pi} e_1 \text{ valid} \quad \dots \quad \Sigma \vdash_{\pi} e_n \text{ valid}}{\Sigma \vdash_{\pi} p(\bar{e}) \text{ valid}} \text{ (CHKPRIM)} \quad \frac{\Sigma \vdash_{\pi} e' \text{ valid}}{\Sigma \vdash_{\pi} v \text{ fby } e' \text{ valid}} \text{ (CHKFBY)} \\
 \frac{\Sigma \vdash \text{rec } x. e \Downarrow^* V \quad \Sigma[x \mapsto V] \vdash_{\pi} e \text{ valid}}{\Sigma \vdash_{\pi} \text{rec } x. e[x] \text{ valid}} \text{ (CHKREC)} \\
 \frac{\Sigma \vdash_{\pi} e \text{ valid} \quad \Sigma \vdash e \Downarrow^* V \quad \Sigma[x \mapsto V] \vdash_{\pi} e' \text{ valid}}{\Sigma \vdash_{\pi} \text{let } x = e \text{ in } e'[x] \text{ valid}} \text{ (CHKLET)} \\
 \frac{(\pi = \pi' \implies \Sigma \vdash e \Downarrow^* \top) \quad \Sigma \vdash_{\pi} e \text{ valid}}{\Sigma \vdash_{\pi} \text{check}_{\pi'} e \text{ valid}} \text{ (CHKCHECK)} \\
 \frac{\begin{array}{c} \Sigma \vdash e_{\text{body}} \Downarrow^* V \\ (\pi = \pi' \implies \Sigma \vdash e_{\text{rely}} \Downarrow^* \top) \\ (\pi = \square \implies \Sigma \vdash e_{\text{rely}} \Downarrow^* \top \implies \Sigma[x \mapsto V] \vdash e_{\text{guar}} \Downarrow^* \top) \\ \Sigma \vdash_{\pi} e_{\text{rely}} \text{ valid} \end{array}}{(\Sigma \vdash e_{\text{rely}} \Downarrow^* \top \implies \Sigma \vdash_{\pi} e_{\text{body}} \text{ valid} \wedge \Sigma[x \mapsto V] \vdash_{\pi} e_{\text{guar}} \text{ valid})} \text{ (CHKCONTRACT)}
 \end{array}$$

Figure 6 Checked semantics for Pipit; the judgment form $\Sigma \vdash_{\pi} e \text{ valid}$ denotes that evaluating expression e under streaming history Σ satisfies the checks and rely-guarantee contract requirements that are labelled with property status π .

Rule CHKLET checks a let-expression `let` $x = e$ `in` e' descends into both sub-expressions. To check the body e' , the rule first evaluates e and extends the streaming environment.

Finally, the heavy lifting is performed by rules CHKCHECK and CHKCONTRACT.

Rule CHKCHECK checks the properties marked π in an expression $\text{check}_{\pi'} e$. If the check-expression has the same status as what we are checking ($\pi = \pi'$), then we evaluate the expression e and require it to be true at all steps. We then unconditionally descend into the subexpression to check any nested properties. Such nested properties are unlikely to be written directly by the user, but might occur after inlining.

Rule CHKCONTRACT applies when checking property status π of a contract with expression $\text{contract}_{\pi'} \{e_{\text{rely}}\} e_{\text{body}} \{x. e_{\text{guar}}[x]\}$. This rule checks both the contract definition and the call-site. We evaluate the body to a stream V ; these values are used to check that the body satisfies guarantee. Although the contract only has one property status, conceptually there are two distinct properties: one for the caller (π') and one for the definition (assumed to be \square). To check the caller property when $\pi = \pi'$, we evaluate the rely e_{rely} and require it to hold. To check the definition property when $\pi = \square$, we assume that the rely holds, and check that the body satisfies the guarantee. We also descend into the subexpressions to check them; when checking the body and guarantee, we can assume that the rely holds.

3.2.1 Blessing expressions and contracts

Blessing is a meta-operation that replaces the property statuses in an expression so that all checks and contracts are marked as valid (\checkmark). Blessing an expression requires a proof that, for all input streams, assuming the valid checks hold, then the unknown checks hold:

$$\frac{\forall \Sigma. \Sigma \vdash_{\square} e \text{ valid} \implies \Sigma \vdash_{\square} e \text{ valid}}{\text{bless } e} \text{ (BLESSEXPRESSION)}$$

We generally prove the required properties by first translating the program to an abstract transition system, as described in Section 4.

Blessing is different for contract definitions, as we need to separate the definition of the contract from the instantiation. To check that a contract definition is valid, we show that if the rely clause is always true for a particular input, then the body satisfies the guarantee for the same inputs. We also assume that the valid properties in the rely, body and guarantee hold, and show the corresponding unknown properties:

$$\begin{aligned} \text{let contract_valid } & \{e_{\text{rely}}\} \{e_{\text{body}}\} \{e_{\text{guar}}\} : \text{prop} = \\ & \forall \Sigma. (\Sigma \vdash_{\square} (e_{\text{rely}}, e_{\text{body}}, e_{\text{guar}}[x := e_{\text{body}}]) \text{ valid} \wedge \Sigma \vdash e_{\text{rely}} \Downarrow^{\square} \top) \\ & \implies (\Sigma \vdash_{\square} (e_{\text{rely}}, e_{\text{body}}, e_{\text{guar}}[x := e_{\text{body}}]) \text{ valid} \wedge \Sigma \vdash e_{\text{guar}}[x := e_{\text{body}}] \Downarrow^{\square} \top) \end{aligned}$$

After proving that the contract is valid for all inputs, we can bless the contract definition. Blessing the contract definition blesses the subexpressions for the rely, body and guarantee, but leaves the contract's *instantiation* property status as unknown:

$$\frac{\text{contract_valid } \{e_{\text{rely}}\} \{e_{\text{body}}\} \{e_{\text{guar}}\}}{\text{bless_contract } \{e_{\text{rely}}\} \{e_{\text{body}}\} \{e_{\text{guar}}\}} \text{ (BLESSCONTRACT)}$$

3.3 Causality and metatheory

To ensure that recursive streams have a computational interpretation, we implement a causality restriction, similar to standard Lustre [10]. This restriction checks that all recursive streams are guarded by a followed-by delay. We implement this as a simple syntactic check: each `rec` $x. e$ can only mention x inside a followed-by. This check ensures productivity of recursive streams, but can be too strict: for example, the expression `rec` $x. (\text{let } x' = x + 1 \text{ in } 0 \text{ fby } x')$ mentions the recursive stream x outside of the delay and is outlawed, but after inlining the let, it would be causal. We hope to relax this restriction in future work.

The causality restriction gives us some important properties about the metatheory. The most important property is that the dynamic semantics form a total function: given a streaming history and a causal expression, we can evaluate the expression to a value. These properties are mechanised in F^* .

► **Theorem 1** (bigstep-is-total). *For any non-empty streaming history Σ and causal expression e , there exists some value v such that e evaluates to v ($\Sigma \vdash e \Downarrow v$).*

The relationship between substitution and the streaming history is also important. In general, we have a substitution property that states that evaluating a substituted expression $e[x := e']$ under some context Σ is equivalent to evaluating e' and adding it to the context Σ :

► **Theorem 2** (bigstep-substitute). *For a streaming history Σ and causal expressions e and e' , if $e[x := e']$ evaluates to a value v ($\Sigma \vdash e \Downarrow v$), then we can evaluate e' to some stream V ($\Sigma \vdash e' \Downarrow^* V$) and extend the streaming history to evaluate e to the original value ($\Sigma[x \mapsto V] \vdash e \Downarrow v$). The converse is also true.*

```

type system (input:  $\Gamma$ ) (result:  $\tau$ ) = {
    state:  $\Gamma$ ;
    free:  $\Gamma$ ;
    init: heap state;
    step: heap input  $\rightarrow$  heap free  $\rightarrow$  heap state  $\rightarrow$  step_result state result;
}

type step_result (state:  $\Gamma$ ) (result:  $\tau$ ) = {
    update: heap state;
    value: result;
    rely: prop;
    guar: prop;
}

```

■ **Figure 7** Abstract transition system type definitions.

The big-step semantics in Figure 5 for a recursive expression `rec` $x. e$ performs one step of recursion by substituting x for the recursive expression. An alternative non-syntax-directed semantics would be to have the environment outside the semantics supply a stream V such that if we extend the streaming history with $x \mapsto V$, then e evaluates to V itself. The above substitution theorem can be used to show that, for causal expressions, these two semantics are equivalent. We can additionally show that, when evaluating e with $x \mapsto V$, the most recent value in V does not affect the result. This fact can be used to “seed” evaluation by starting with an arbitrary value:

► **Theorem 3** (bigstep-rec-causal). *For a streaming history $\Sigma; \sigma$ and a causal recursive expression `rec` $x. e$, if $(\Sigma; \sigma \vdash e \Downarrow v)$, then updating $\sigma[x]$ with any value v' results in the same value: $(\Sigma; \sigma[x \mapsto v'] \vdash e \Downarrow v)$.*

4 Abstract transition systems

To prove properties about Pipit programs, we translate to an *abstract* transition system, so-called because it abstracts away the implementation details of contract instantiations. For extraction we also translate to *executable* transition systems, which we discuss in Section 5.

Figure 7 shows the types of transition systems. A transition system is parameterised by its input context and the result type. It also contains two internal contexts: firstly, the state context describes the private state required to execute the machine; secondly, the free context contains any extra input values that the transition system would like to existentially quantify over. The free context is used to allow the system to ask for arbitrary values from the environment, when it would not otherwise be able to return a concrete value.

For recursive streams and contract instantiations, which hide their implementation, the natural translation to a transition system would involve existentially quantifying a result that satisfies the specification. Unfortunately, using an existential quantifier requires a step *relation* rather than a step *function*. Using a step relation complicates the resulting transition system, as other operations such as primitive application must also introduce existential quantifiers; such quantifiers block simplifications such as partial-evaluation and result in a more complex transition system. Instead, the free context provides the step function with a fresh unconstrained value of the desired type, which the step function can then constrain.

Back to Figure 7, the step-result contains the updated state for the transition system, as well as the result value. The step-result additionally contains two propositions; one for the “rely”, or assumptions about the execution environment, and another for the “guarantee”, or obligations that the transition system must show. For the transition system corresponding to an expression e , these propositions are roughly analogous to the known checked semantics $\Sigma \vdash \square e$ valid and unknown checks $\Sigma \vdash ? e$ valid respectively.

For example, recall again the sum contract:

```
let sum ints = contract[?] {ints > 0} (rec sum. (0 fby sum) + ints) {sum.sum > 0}
```

To verify the contract definition, we first translate it to an abstract transition system whose input environment contains an integer $ints$, and whose result type is also an integer. The followed-by delay results in a local state variable called sum_fby , and we encode the existentially-quantified recursive stream as a free context variable called sum :

```
let sum_def: system (ints: Z) Z = {
    state   = (sum_fby: Z);
    free    = (sum: Z);
    init    = { sum_fby = 0 };
    step    = λi f s. {
        update = { sum_fby = f.sum };
        value  = f.sum;
        rely   = (f.sum = s.sum_fby + i.ints) ∧ i.ints > 0;
        guar   = f.sum > 0; } }
```

The initial state of 0 corresponds to the initial value of the followed-by. In the step function, argument i refers to the input heap containing $i.ints$, f refers to the free heap containing the recursive stream $f.sum$, and s refers to the state heap containing $s.sum_fby$. In the rely of the step result, $f.sum$ is constrained to be the translated body of the recursive stream. The translated rely also includes the contract’s rely that the input integer is positive. Finally, the translated guarantee includes the contract’s guarantee that the output is positive.

To verify the transition system, we prove inductively that if the rely always holds, then the guarantee holds; we discuss proofs of system validity further in Subsection 4.2.

The translation for contract instantiations is similar, except that the contract body is replaced by an arbitrary value from the free context. For example, we can use the sum contract to implement the Fibonacci sequence with $rec fib.sum (1 fby fib)$. This program does not require any input values, so we leave the input context empty. The state context includes an entry for the $1 fby fib$ followed-by expression, but does not include the followed-by expressions inside the contract definition. Similarly, the free context includes an entry for the recursive stream, and an entry for the abstract, underspecified value of the contract:

```
let fib_def: system () Z = {
    state   = (fib_fby: Z);
    free    = (fib: Z; sum_contract: Z);
    init    = { fib_fby = 1 };
    step    = λi f s. {
        update = { fib_fby = f.fib };
        value  = f.fib;
        rely   = (f.fib = f.sum_contract)
                ∧ (s.fib_fby > 0 ⇒ f.sum_contract > 0);
        guar   = s.fib_fby > 0; } }
```

$$\begin{aligned}
\llbracket v \rrbracket_{\text{state}} &= \cdot \\
\llbracket x \rrbracket_{\text{state}} &= \cdot \\
\llbracket p(\bar{e}) \rrbracket_{\text{state}} &= \bigcup_i \llbracket e_i \rrbracket_{\text{state}} \\
\llbracket v \text{ fby } e \rrbracket_{\text{state}} &= x_{\text{fby}(e)} : \tau, \llbracket e \rrbracket_{\text{state}} \quad (\text{fresh } x_{\text{fby}(e)}) \\
\llbracket \text{rec } x. e \rrbracket_{\text{state}} &= \llbracket e \rrbracket_{\text{state}} \\
\llbracket \text{let } x = e \text{ in } e' \rrbracket_{\text{state}} &= \llbracket e \rrbracket_{\text{state}} \cup \llbracket e' \rrbracket_{\text{state}} \\
\llbracket \text{check}_\pi e \rrbracket_{\text{state}} &= \llbracket e \rrbracket_{\text{state}} \\
\llbracket \text{contract}_\pi \{e_r\} e_b \{x. e_g\} \rrbracket_{\text{state}} &= \llbracket e_r \rrbracket_{\text{state}} \cup \llbracket e_b \rrbracket_{\text{state}}
\end{aligned}$$

$$\begin{aligned}
\llbracket v \rrbracket_{\text{free}} &= \cdot \\
\llbracket x \rrbracket_{\text{free}} &= \cdot \\
\llbracket p(\bar{e}) \rrbracket_{\text{free}} &= \bigcup_i \llbracket e_i \rrbracket_{\text{free}} \\
\llbracket v \text{ fby } e \rrbracket_{\text{free}} &= \llbracket e \rrbracket_{\text{free}} \\
\llbracket \text{rec } x. e \rrbracket_{\text{free}} &= x : \tau, \llbracket e \rrbracket_{\text{free}} \\
\llbracket \text{let } x = e \text{ in } e' \rrbracket_{\text{free}} &= \llbracket e \rrbracket_{\text{free}} \cup \llbracket e' \rrbracket_{\text{state}} \\
\llbracket \text{check}_\pi e \rrbracket_{\text{free}} &= \llbracket e \rrbracket_{\text{free}} \\
\llbracket \text{contract}_\pi \{e_r\} e_b \{x. e_g\} \rrbracket_{\text{free}} &= x : \tau, \llbracket e_r \rrbracket_{\text{free}} \cup \llbracket e_b \rrbracket_{\text{state}}
\end{aligned}$$

Figure 8 Transition system typing contexts of expressions; for an expression e , $\llbracket e \rrbracket_{\text{state}} : \Gamma$ and $\llbracket e \rrbracket_{\text{free}} : \Gamma$ describe the heaps used to store the expression’s internal state and extra inputs.

As before, the translated rely includes the assumption that the recursive stream’s value ($f.\text{fib}$) agrees with its body ($f.\text{sum_contract}$). Additionally, the rely includes the assumption that the contract’s rely implies the guarantee: if sum’s input ($s.\text{fib_fby}$) is positive, then its output ($f.\text{sum_contract}$) is positive too. Finally, the translated guarantee encodes the obligation that the environment satisfies the *contract’s rely* – the input to sum is positive.

Note that the transition system requires the rely to hold *at the current step*, while the “true” semantics of contracts requires the rely to hold *at every step so far*. This minor optimisation is sound, as we define system validity to require all steps to satisfy the rely.

4.1 Translation

We now present the details of the translation. For causal expressions, the translated transition system is verified to be an abstraction of the original expression’s dynamic semantics, and the generated proof obligations imply that the original expression satisfies the checked semantics.

Figure 8 defines the internal state and free contexts required for an expression. For most expression forms, the state and free contexts are defined by taking the union of the contexts of subexpressions. Followed-by delays introduce a local state variable $x_{\text{fby}(e)}$ in which to store the most recent stream value. We generate a fresh variable here, although the implementation uses de Bruijn indices. Recursive streams and contracts both introduce new bindings into the free context; we assume that their binders x are unique.

Figure 9 defines the translation for expressions. Values and variables have no internal state. For variables, we look for the variable binding in either of the input or free heaps; bindings are unique and cannot occur in both. We omit the rely and guarantee definitions here; both are trivially true.

To translate primitives, we union together the initial states of the subexpressions; updating the state is similar. For the rely and guarantee definitions, we take the conjunction: we can assume that all subexpressions rely clauses hold, and must show that all guarantees hold.

$\llbracket v \rrbracket_{\text{init}}$	$= ()$
$\llbracket v \rrbracket_{\text{value}}(i, f, s)$	$= v$
$\llbracket x \rrbracket_{\text{init}}$	$= ()$
$\llbracket x \rrbracket_{\text{value}}(i, f, s)$	$= (i \cup f).x$
$\llbracket p(\bar{e}) \rrbracket_{\text{init}}$	$= \bigcup_i \llbracket e_i \rrbracket_{\text{init}}$
$\llbracket p(\bar{e}) \rrbracket_{\text{value}}(i, f, s)$	$= \text{prim-sem}(p, \overline{\llbracket e \rrbracket_{\text{value}}(i, f, s)})$
$\llbracket p(\bar{e}) \rrbracket_{\text{update}}(i, f, s)$	$= \bigcup_i \llbracket e_i \rrbracket_{\text{update}}(i, f, s)$
$\llbracket p(\bar{e}) \rrbracket_{\text{rely}}(i, f, s)$	$= \bigwedge_i \llbracket e_i \rrbracket_{\text{rely}}(i, f, s)$
$\llbracket p(\bar{e}) \rrbracket_{\text{guar}}(i, f, s)$	$= \bigwedge_i \llbracket e_i \rrbracket_{\text{guar}}(i, f, s)$
$\llbracket v \mathbf{fby} e \rrbracket_{\text{init}}$	$= \llbracket e \rrbracket_{\text{init}} \cup \{x_{\mathbf{fby}(e)} \mapsto v\}$
$\llbracket v \mathbf{fby} e \rrbracket_{\text{value}}(i, f, s)$	$= s.x_{\mathbf{fby}(e)}$
$\llbracket v \mathbf{fby} e \rrbracket_{\text{update}}(i, f, s)$	$= \llbracket e \rrbracket_{\text{update}}(i, f, s) \cup \{x_{\mathbf{fby}(e)} \mapsto \llbracket e \rrbracket_{\text{value}}(i, f, s)\}$
$\llbracket v \mathbf{fby} e \rrbracket_{\text{rely}}(i, f, s)$	$= \llbracket e \rrbracket_{\text{rely}}(i, f, s)$
$\llbracket v \mathbf{fby} e \rrbracket_{\text{guar}}(i, f, s)$	$= \llbracket e \rrbracket_{\text{guar}}(i, f, s)$
$\llbracket \text{rec } x. e \rrbracket_{\text{init}}$	$= \llbracket e \rrbracket_{\text{init}}$
$\llbracket \text{rec } x. e \rrbracket_{\text{value}}(i, f, s)$	$= f.x$
$\llbracket \text{rec } x. e \rrbracket_{\text{update}}(i, f, s)$	$= \llbracket e \rrbracket_{\text{update}}(i, f, s)$
$\llbracket \text{rec } x. e \rrbracket_{\text{rely}}(i, f, s)$	$= \llbracket e \rrbracket_{\text{rely}}(i, f, s)$
$\llbracket \text{rec } x. e \rrbracket_{\text{guar}}(i, f, s)$	$\wedge f.x = \llbracket e \rrbracket_{\text{value}}(i, f, s)$ $= \llbracket e \rrbracket_{\text{guar}}(i, f, s)$
$\llbracket \text{let } x = e \text{ in } e' \rrbracket_{\text{init}}$	$= \llbracket e \rrbracket_{\text{init}} \cup \llbracket e' \rrbracket_{\text{init}}$
$\llbracket \text{let } x = e \text{ in } e' \rrbracket_{\text{value}}(i, f, s)$	$= \llbracket e' \rrbracket_{\text{value}}(i \cup \{x \mapsto \llbracket e \rrbracket_{\text{value}}(i, f, s)\}, f, s)$
$\llbracket \text{let } x = e \text{ in } e' \rrbracket_{\text{update}}(i, f, s)$	$= \llbracket e' \rrbracket_{\text{update}}(i \cup \{x \mapsto \llbracket e \rrbracket_{\text{value}}(i, f, s)\}, f, s)$ $\cup \llbracket e \rrbracket_{\text{update}}(i, f, s)$
$\llbracket \text{let } x = e \text{ in } e' \rrbracket_{\text{rely}}(i, f, s)$	$= \llbracket e' \rrbracket_{\text{rely}}(i \cup \{x \mapsto \llbracket e \rrbracket_{\text{value}}(i, f, s)\}, f, s)$ $\wedge \llbracket e \rrbracket_{\text{rely}}(i, f, s)$
$\llbracket \text{let } x = e \text{ in } e' \rrbracket_{\text{guar}}(i, f, s)$	$= \llbracket e' \rrbracket_{\text{guar}}(i \cup \{x \mapsto \llbracket e \rrbracket_{\text{value}}(i, f, s)\}, f, s)$ $\wedge \llbracket e \rrbracket_{\text{guar}}(i, f, s)$
$\llbracket \text{check}_\pi e \rrbracket_{\text{init}}$	$= \llbracket e \rrbracket_{\text{init}}$
$\llbracket \text{check}_\pi e \rrbracket_{\text{value}}(i, f, s)$	$= ()$
$\llbracket \text{check}_\pi e \rrbracket_{\text{update}}(i, f, s)$	$= \llbracket e \rrbracket_{\text{update}}(i, f, s)$
$\llbracket \text{check}_\pi e \rrbracket_{\text{rely}}(i, f, s)$	$= (\pi = \square \implies \llbracket e \rrbracket_{\text{value}}(i, f, s)) \wedge \llbracket e \rrbracket_{\text{rely}}(i, f, s)$
$\llbracket \text{check}_\pi e \rrbracket_{\text{guar}}(i, f, s)$	$= (\pi = \square \implies \llbracket e \rrbracket_{\text{value}}(i, f, s)) \wedge \llbracket e \rrbracket_{\text{guar}}(i, f, s)$
$\llbracket \text{contract}_\pi \{e_r\} e_b \{x. e_g\} \rrbracket_{\text{init}}$	$= \llbracket e_r \rrbracket_{\text{init}} \cup \llbracket e_g \rrbracket_{\text{init}}$
$\llbracket \text{contract}_\pi \{e_r\} e_b \{x. e_g\} \rrbracket_{\text{value}}(i, f, s)$	$= f.x$
$\llbracket \text{contract}_\pi \{e_r\} e_b \{x. e_g\} \rrbracket_{\text{update}}(i, f, s)$	$= \llbracket e_r \rrbracket_{\text{update}}(i, f, s) \cup \llbracket e_g \rrbracket_{\text{update}}(i, f, s)$
$\llbracket \text{contract}_\pi \{e_r\} e_b \{x. e_g\} \rrbracket_{\text{rely}}(i, f, s)$	$= (\llbracket e_r \rrbracket_{\text{value}}(i, f, s) \implies \llbracket e_g \rrbracket_{\text{value}}(i, f, s))$ $\wedge (\pi = \square \implies \llbracket e_r \rrbracket_{\text{value}}(i, f, s))$ $\wedge \llbracket e_r \rrbracket_{\text{rely}}(i, f, s)$
$\llbracket \text{contract}_\pi \{e_r\} e_b \{x. e_g\} \rrbracket_{\text{guar}}(i, f, s)$	$= (\llbracket e_r \rrbracket_{\text{value}}(i, f, s) \implies \llbracket e_g \rrbracket_{\text{rely}}(i, f, s))$ $= (\pi = \square \implies \llbracket e_r \rrbracket_{\text{value}}(i, f, s))$ $\wedge \llbracket e_r \rrbracket_{\text{guar}}(i, f, s) \wedge \llbracket e_g \rrbracket_{\text{guar}}(i, f, s)$

Figure 9 Transition system semantics; for an expression $\Gamma \vdash e : \tau$, $\llbracket e \rrbracket_{\text{init}}$ is the initial state. For each field of the step-result type, we define a translation function that takes the input, free and state heaps: for example, we define the value-result of a step with type $\llbracket e \rrbracket_{\text{value}} : \text{heap } \Gamma \rightarrow \text{heap } \llbracket e \rrbracket_{\text{free}} \rightarrow \text{heap } \llbracket e \rrbracket_{\text{state}} \rightarrow \tau$.

To translate a followed-by $v \text{ fby } e$, we initialise the followed-by's unique binder $x_{\text{fby}(e)}$ to the followed-by's default value v . At each step, we return the value in the local state before updating the local state to the subexpression's new value.

To translate a recursive expression $\text{rec } x. e$ of type τ , we require an arbitrary value $x : \tau$ in the free heap. The rely proposition constrains the free variable x to be the result of evaluating e with the binding for x passed along, thus closing the recursive loop.

To translate let-expressions $\text{let } x = e \text{ in } e'$, we extend the input heap with the value of e before evaluating e' . The presentation here duplicates the computation of the value of e , but the actual implementation introduces a single binding.

To translate a check property, we inspect the property status. If the property is known to be valid, then we can assume the property is true in the rely clause. Otherwise, we include the property as an obligation in the guarantee clause. In either case, we also include the subexpression's rely and guarantee clauses.

Finally, to translate contract instantiations, we use the contract's rely and guarantee and ignore the body. As with recursive expressions, we require an arbitrary value $x : \tau$ in the free heap. The translation's rely allows us to assume that the contract definition holds: that is, the contract's rely implies the contract's guarantee. If the contract instantiation is known to be valid, we can also assume that the contract's rely holds. Otherwise, we include the contract's rely as an obligation by putting it in the translation's guarantee.

4.2 Proof obligations and induction

To verify that the translated system satisfies its proof obligations – that is, the checked properties and contract relies hold – we can perform induction on the system's sequence of steps. A system satisfies its proof obligations if, for any sequence of steps that all satisfy its rely or assumptions, the system's guarantee also holds for all of the steps.

Inductive proofs on Lustre programs generally use a non-standard definition of induction, as the property we wish to show is a function of the *step result*, rather than being a function of the *state*. This means that the base case must take a single step from the initial state to be able to state the property that, if the step result's rely holds, then its guarantee holds:

```
let inductive_check_base (sys : system input  $\tau$ ) : prop =
   $\forall(i : \text{heap input})(f : \text{heap sys.free}).$ 
  let stp = sys.step i f sys.init in
  stp.rely  $\implies$  stp.guar
```

For the inductive step case, we allow the system to take *two* steps from an arbitrary state, assuming that both steps satisfy the rely and the first step satisfied the inductive property:

```
let inductive_check_step (sys : system input  $\tau$ ) : prop =
   $\forall(i_0 i_1 : \text{heap input})(f_0 f_1 : \text{heap sys.free})(s_0 : \text{heap sys.state}).$ 
  let stp1 = sys.step i0 f0 s0 in
  let stp2 = sys.step i1 f1 stp1.state in
  stp1.rely  $\implies$  stp1.guar  $\implies$  stp2.rely  $\implies$  stp2.guar
```

This inductive scheme also generalises to *k-induction*, which allows the inductive case to assume the previous *k* steps satisfied the inductive property, rather than just assuming that the one previous step holds. K-induction is a fairly standard invariant strengthening technique; intuitively, it allows the proof to use more context of the history of execution [21, 11, 16].

To reason about system validity in general, we define a predicate *system_holds_all* that formally defines a valid system as: for all sequences of inputs and their corresponding steps, if all of the steps' relies hold, then the guarantees also hold. Validity is implied by (k-)induction.

$$\begin{array}{c}
 \boxed{\Sigma \vdash e \sim s} \\
 \frac{}{\Sigma \vdash v \sim s} \text{ (IVALE) } \qquad \qquad \frac{}{\Sigma \vdash x \sim s} \text{ (IVAR) } \\
 \frac{\Sigma \vdash e_1 \sim s \quad \dots \quad \Sigma \vdash e_n \sim s}{\Sigma \vdash p(\bar{e}) \sim s} \text{ (IPRIM) } \qquad \qquad \frac{s.x \mathbf{fby}(e') = v \quad \cdot \vdash e' \sim s}{\cdot \vdash v \mathbf{fby} e' \sim s} \text{ (IFBY}_0\text{) } \\
 \frac{\Sigma; \sigma \vdash e' \Downarrow s.x \mathbf{fby}(e') \quad \Sigma; \sigma \vdash e' \sim s}{\Sigma; \sigma \vdash v \mathbf{fby} e' \sim s} \text{ (IFBY}_S\text{) } \\
 \frac{\Sigma \vdash \mathbf{rec} \ x. e \Downarrow^* V \quad \Sigma[x \mapsto V] \vdash e \sim s}{\Sigma \vdash \mathbf{rec} \ x. e[x] \sim s} \text{ (IREC) } \\
 \frac{\Sigma \vdash e \Downarrow^* V \quad \Sigma \vdash e \sim s \quad \Sigma[x \mapsto V] \vdash e' \sim s}{\Sigma \vdash \mathbf{let} \ x = e \ \mathbf{in} \ e'[x] \sim s} \text{ (ILET) } \\
 \frac{\Sigma \vdash e \sim s}{\Sigma \vdash \mathbf{check}_\pi e \sim s} \text{ (ICHECK) } \\
 \frac{\Sigma \vdash e_{\text{body}} \Downarrow^* V \quad \Sigma \vdash e_{\text{rely}} \sim s \quad \Sigma[x \mapsto V] \vdash e_{\text{guar}} \sim s}{\Sigma \vdash \mathbf{contract}_\pi \{e_{\text{rely}}\} e_{\text{body}} \{x. e_{\text{guar}}[x]\} \sim s} \text{ (ICONTRACT) }
 \end{array}$$

Figure 10 Transition system state invariant.

4.3 Translation correctness proofs

We prove that the transition system is an abstraction of the dynamic semantics: that is, if the expression evaluates to v under some context, then there exists some execution of the transition system that also results in v . The transition system itself is deterministic, but the free context provides the non-determinism which may occur from underspecified contracts; our theorem statement existentially quantifies the free heap.

The results presented here rely heavily on the totality and substitution metaproPERTIES described in Subsection 3.3. Figure 10 defines the invariant for the abstraction proof; the judgment form $\Sigma \vdash e \sim s$ checks that s is a valid state heap. We use the invariant to state that, if executing the transition system for e on the entire streaming history Σ results in state heap s , then s is a valid state.

As most expressions do not modify the state heap, the invariant for most expressions simply descends into the subexpressions. Where new bindings are added, we use the dynamic semantics to extend the context with the new values. The invariant for followed-by expressions asserts that the initial state of the followed-by is the default value; on subsequent steps, the state corresponds to the dynamic semantics. With this invariant, we can prove abstraction:

► **Theorem 4** (translation-abstraction). *For a well-typed causal expression e and streaming history Σ , if e evaluates to stream V ($\Sigma \vdash e \Downarrow^* V$), then there exists a sequence of free heaps Σ_F such that repeated application of the transition system's step results in V .*

Finally, we can show the main entailment result that if the proof obligations hold on the system, then the original program is valid according to the checked semantics:

► **Theorem 5** (translation-entailment). *For a well-typed causal expression e and its translated system s , if the system holds ($\text{system_holds_all } s$), and the checked properties in e hold ($\forall \Sigma. \Sigma \vdash \square e \text{ valid}$), then the unknown properties in e also hold ($\forall \Sigma. \Sigma \vdash \square? e \text{ valid}$)*

The above theorem allows us to *bless* the expression and mark all properties as valid (Subsubsection 3.2.1). Importantly, the assumption that the checked properties hold lets us re-use previously-verified properties without re-proving them, allowing for modular proofs.

5 Extraction

Pipit can generate executable code which is suitable for real-time execution on embedded devices. The code extraction uses a variation of the abstract transition system described in Section 4, with two main differences to ensure that the result is executable without relying on the environment to provide values for the free context. Contracts are straightforward to execute by using the body of the contract rather than abstracting over the implementation.

To execute recursive expressions $\text{rec } x. e : \tau$, we require an arbitrary value of type τ to seed the fixpoint, as described in Subsection 3.3. We first call the step function to evaluate e with x bound to \perp_τ . This step call returns the correct value, but the updated state is invalid, as it may refer to the bottom value. To get the correct state, we call the step function again, this time with x bound to the correct value, v .

For example, for the *sum* contract with body $(\text{rec } sum. (0 \text{ fby } sum) + ints)$, we generate an executable system that takes an input context containing integer variable *ints*, with a single state variable for the followed-by, and returning an integer:

```
let sum_def: system (ints:  $\mathbb{Z}$ )  $\mathbb{Z} = \{$ 
    state = (sum_fby:  $\mathbb{Z}$ );
    init = { sum_fby = 0; };
    step =  $\lambda i s.$ 
        let (fby0, s0) = (s.sum_fby, s {sum_fby =  $\perp_{\mathbb{Z}}$ }) in
        let (sum0, s0) = (fby0 + i.ints, s0) in
        let (fby1, s1) = (s.sum_fby, s {sum_fby = sum0}) in
        let (sum1, s1) = (fby1 + i.ints, s1) in
        (sum0, s1) }
```

Here, the step function takes heaps of the input and state contexts, and returns a pair of the result value and the updated state. The first two bindings correspond to the seeded evaluation with the recursive value for the sum set to $\perp_{\mathbb{Z}}$; as such, the resulting state s_0 is invalid. The last two bindings recompute the state, this time with the correct recursive value sum_0 used in the state. This duplication of work can often be removed by the partial evaluation and dead-code-elimination which we perform during code extraction.

This translation to transition systems is verified to preserve the original semantics. The invariant is very similar to that of Subsection 4.3, except that the invariant descends into the implementations of contracts. For the abstract systems we only showed abstraction; to prove that executable systems are equivalent to the original semantics, we use the fact that the original semantics and transition systems are both deterministic and total (Subsection 3.3).

► **Theorem 6** (execution-equivalence). *For a well-typed causal expression e and streaming history Σ , e evaluates to stream V ($\Sigma \vdash e \Downarrow^* V$) if-and-only-if repeated application of the transition system's step on Σ also results in V .*

To extract the program, we use a *hybrid embedding* as described in [23], which is similar to staged-compilation. The hybrid embedding involves a deep embedding of the Pipit core language, while the translation to executable transition systems produces a shallow embedding. We use the F^{*} host language's normalisation-by-evaluation and tactic support [31] to partially-evaluate the application of the translation to a particular input program. This partial-evaluation results in a concrete transition system that fits in the Low^{*} subset of F^{*}, which can then be extracted to statically-allocated C code [34].

The generated C code for *sum*² includes a struct type to hold the state information, as well as reset and step functions:

```
struct sum_state { uint32_t sum_fby; }
void sum_reset(struct sum_state* state);
int sum_step(struct sum_state* state, uint32_t ints);
```

The reset function takes the pointer to the state struct and sets it to its initial values. The step function takes the pointer to the state struct and the inputs, and returns the result integer. The state struct is updated in-place. The implementations of these functions avoid dynamic (heap) allocation and are suitable for embedded systems. This interface is standard for Lustre compilers [5, 19] and other synchronous languages.

Unfortunately, our current approach is unsuitable for generating imperative array code, as our pure transition system only supports pure arrays. In the future, we intend to support efficient array computations and fix the above work duplication by introducing an intermediate imperative language such as Obc [3], a static object-based language suitable for synchronous systems. Even with an added intermediate language, we believe that a variant of our current translation and proof-of-correctness will remain useful as an intermediate semantics.

6 Evaluation

To evaluate Pipit, we have implemented the high-level logic of a Time-Triggered Controller Area Network (TTCAN) bus driver [1], described earlier in Section 2. The CAN bus is common in safety-critical automotive and industrial settings. The time-triggered network architecture defines a static schedule of network traffic; by having all nodes on the network adhere to the schedule, the reliability of periodic messages is significantly increased [15].

The TTCAN protocol can be implemented in two levels of increasing complexity. In the first level, reference messages, which perform synchronisation between nodes, contain the index of the newly-started cycle. In the second level, the reference messages also contain the value of a global fractional clock and whether any gaps have occurred in the global clock, which allows other nodes to calibrate their own clocks. We implement the first level as it is more amenable to software implementation [22].

The implementation defines a streaming function that takes a stream describing the current time, the state of the hardware, and any received messages. It returns a stream of commands to be performed, such as sending a particular reference message. The implementation defines a pure streaming function. To actually interact with the hardware we assume a small hardware-interop layer that reads from the hardware registers and translates the commands to hardware-register writes, but we have not yet implemented this. We package the driver's inputs into a record for convenience:

² This interface is for a variant of the sum contract with 32-bit integers instead of unbounded integers.

```
type driver_input = {
    local_time: network_time_unit;
    mode_cmd: option mode;
    tx_status: tx_status;
    bus_status: bus_status;
    rx_ref: option ref_message;
    rx_app: option app_message_index;
}
```

Here, the local-time field denotes the time-since-boot in *network time units*, which are based on the bitrate of the underlying network bus. The mode-command is an optional field which indicates requests from the application to enter configuration or execution mode. The transmission-status describes the status of the last transmission request and may be none, success, or various error conditions. The bus-status describes whether the bus is currently idle, busy, or in an error state. The two receive fields denote messages received from the bus; for application-specific messages the time-triggered logic only needs the message identifier.

The driver-logic returns a stream of commands for the hardware-interop layer to perform:

```
type commands = {
    enable_acks: bool;
    tx_ref: option ref_message;
    tx_app: option app_message_index;
    tx_delay: network_time_unit;
}
```

The enable-acknowledgements field denotes whether the hardware should respond to messages from other nodes with an acknowledgement bit; in the case of a severe error acknowledgements are disabled, as the node must not write to the bus at all. The transmit fields denote whether to send a reference message or an application-specific message. For application-specific messages, the hardware-interop layer maintains the transmission buffers containing the actual message payload. To meet the schedule as closely as possible, the driver anticipates the next transmission and includes a transmission delay to tell the hardware exactly when to send the next message.

6.1 Runtime

The implementation includes an extension of the trigger-fetch logic described in Section 2, as well as state machines for tracking node synchronisation, master status and fault handling. We generate real-time C code as described in Section 5. We evaluated the generated C code by executing with randomised inputs and measuring the worst-case-execution-time on a Raspberry Pi Pico (RP2040) microcontroller. The runtime of the driver logic is fairly stable: over 5,000 executions, the measured worst-case execution time was $140\mu s$, while the average was $90\mu s$ with a standard deviation of $1.5\mu s$. Earlier work on fault-tolerant TTCAN [41] describes the required slot sizes – the minimum time between triggers – to achieve bus utilisation at different bus rates. For a 125Kbit/s bus, a slot size of approximately $1,500\mu s$ is required to achieve utilisation above 85 per cent. For the maximum CAN bus rate of 1Mbit/s, the required slot size is $184\mu s$. Further evaluation is required to ensure that the complete runtime including the hardware-interop layer is sufficient for full-speed CAN.

Our code generation can be improved in a few ways. A common optimisation in Lustre is to fuse consecutive if-statements with the same condition [5]; such an optimisation seems useful here, as our treatment of optional values introduces repeated unpacking and repacking.

```

function next(index: int; c: cycle)
    returns (result: int)
var next_array: int ^ COUNT;
let
    next_array[i] =
        if trigger_enabled(COUNT - 1 - i, c)
        then COUNT - 1 - i
        else if i <= 0
        then NO_NEXT_TRIGGER
        else next_array[i - 1];
    result =
        next_array[COUNT - 1 - index];
tel

```

Figure 11 Left: next-trigger logic in F^{*}; right: Kind2 encoding as array scan. In F^{*}, the *Tot* τ (*decreases* ...) syntax declares a total function with the given termination measure. In Kind2, the *int*[^]*COUNT* syntax denotes the type of an array of integers of length *COUNT*, while the *next_array*[*i*] declaration defines the elements of the array as a function of the index *i*.

Some form of array fusion [37] may also be useful for removing redundant array operations. Our current extraction generates a transition-system with a step function which returns a tuple of the updated state and result. Composing these step functions together results in repeated boxing and unboxing of this tuple; we currently rely on the F^{*} normaliser to remove this boxing. In the future, we plan to build on the current proofs to implement a more-sophisticated encoding that introduces less overhead.

6.2 Verification

We have verified a simplified trigger-fetch mechanism, as presented earlier (Section 2). For comparison, we implemented the same logic in the Kind2 model-checker [11]. The restrictions placed on the triggers array – that triggers are sorted by time-mark, that there must be an adequate time-gap between a trigger and its next-enabled, and that a trigger’s time-mark must be greater-than-or-equal-to its index – are naturally expressed with quantifiers. The Kind2 model-checker includes experimental array and quantifier support [26]. Due to the experimental nature of these features, we had to work around some limitations: for example, the use of arrays and quantifiers disables IC3-based invariant generation; quantified variables cannot be used in function calls; and the use of top-level constant arrays caused runtime errors that rendered most properties invalid [27].

We were able to express equivalent properties in Kind2 and in Pipit, aside from some encoding issues. For example, the specification-only function that finds the next trigger is naturally recursive. Kind2 does not support recursive functions, but we were able to encode it by introducing a temporary array and using Kind2’s array comprehension syntax for scanning over arrays. Additionally, while the recursive call *increases* the index, the array scan can only depend on values with lower indices. Figure 11 illustrates this encoding with a simplified version of the next-trigger logic.

We compare against two Kind2 implementations: one corresponds closely to the Pipit development, while the other includes a critical simplification to modify the trigger-enabled set to be a single cycle index. In TTCAN proper, the enabled set is implemented as a cycle-offset and repeat-factor. Checking if a trigger is enabled in the current cycle requires

size	Kind2				Pipit	
	simple enable-set		full enable-set		wall-clock	CPU time
	wall-clock	CPU time	wall-clock	CPU time		
1	1.48s	1.06s	1.57s	2.26s	5.25s	5.03s
2	1.51s	1.26s	1.71s	2.93s	5.25s	5.03s
4	1.57s	1.62s	2.08s	4.78s	5.25s	5.03s
8	1.76s	3.07s	4.21s	16.98s	5.25s	5.03s
16	3.36s	11.91s	13.82s	65.57s	5.25s	5.03s
32	12.15s	62.38s	269.14s	1230.05s	5.25s	5.03s
64	1701.01s	9096.99s	(timeout)		5.25s	5.03s
128	(timeout)		(timeout)		5.25s	5.03s

Figure 12 Verification time for trigger-fetch; simple enable-set uses a simplified version of the enable-set, while full enable-set uses bitwise arithmetic as in the TTCAN specification. The wall-clock time denotes the elapsed time that an engineer must spend waiting for the result; the CPU time denotes the total time spent computing by all of the CPU cores. The verification time for Pipit is a once-and-for-all proof that is parametric in the size of the array. The time limit was one hour.

nonlinear arithmetic, which is difficult for SMT solvers. In our Pipit development, we can treat the definition of the cycle set abstractly. However, in the Kind2 development, quantified formulas cannot contain function calls, which means that we cannot hide the implementation of the enabled-set check by providing an abstract contract. This limitation also makes the specification quite unwieldy, as we must manually inline any functions in quantified formulas.

Figure 12 shows the verification runtime for different sizes of arrays; the Pipit version is parametric in the array size, and is thus verified for all sizes of arrays. We ran these experiments in Docker on an Intel i5-12500 with 32GB of RAM. Both Kind2 and Pipit developments of the trigger-fetch logic are roughly the same size, on the order of two-hundred lines of code including comments. Ignoring whitespace and comments, the Pipit implementation of trigger-fetch has 26 lines of actual executable code, while the Kind2 code has 32. The majority of the remaining code comprises the definition of valid schedules (34 for Pipit, 28 for Kind2), and the lemma statements and invariants (12 for Pipit, 31 for Kind2), as well as contract statements and boilerplate.

We were able to verify the Kind2 implementation of the complete trigger-fetch mechanism for up to 32 triggers; above that, our verification timed out after one hour. For the simplified trigger-fetch mechanism, we were able to verify up to 64 triggers. For reference, hardware implementations of TTCAN such as M_TTCAN support up to 64 triggers [36].

We plan to verify the remainder of the TTCAN implementation and publish it separately. Prior work formalising TTCAN has variously modeled the protocol itself [39, 33, 30], instances of the protocol [20], and abstract models of TTCAN implementations [29], but we are unaware of any prior work that has verified an *executable* implementation of TTCAN.

Separately, Pipit has also been used to implement and verify a real-time controller for a coffee machine reservoir control system [38]. The reservoir has a float switch to sense the water level and a solenoid to allow the intake of water. The specification includes a simple model of the water reservoir and shows that the reservoir does not exceed the maximum level under different failure-mode assumptions.

7 Related work

Using existing Lustre tools to verify *and* execute the time-triggered CAN driver from Section 2 is nontrivial. Compiling the triggers array with an unverified compiler such as Lustre V6 [24] or Heptagon [19] is straightforward; however, the verified Lustre compiler Vélus [7] does not support arrays, records, or a foreign-function interface. Recent work on translation validation for LustreC [9] also does not yet support arrays.

Verifying the time-triggered CAN driver is trickier, as the restrictions placed on the triggers array – that triggers are sorted by time-mark, there must be an adequate time-gap between a trigger and its next-enabled, and a trigger’s time-mark must be greater-than-or-equal-to its index – naturally require quantifiers. As described in Section 6, Kind2 does include experimental array and quantifier support, but in our experiments was unable to verify the full logic for arrays up to the 64 triggers, which is the size supported by hardware implementations of TTCAN. Additionally, due to the limitations that require the constant triggers array to be passed as an argument, compiling the program with Lustre V6 would result in the entire triggers array being copied to the stack each iteration, which is unlikely to result in acceptable performance.

Other model-checkers for Lustre such as Lesar [35], JKind [16] and the original Kind [21] do not support quantifiers. It may be possible to encode the quantifiers as fixed-size loops in those that support arrays, but ensuring that these loops do not affect the execution or runtime complexity of the generated code does not appear to be straightforward.

These model-checkers have definite usability advantages over the general-purpose-prover approach offered here: they can often generate concrete counterexamples and implement counterexample-based invariant-generation techniques such as ICE [18] and PDR [8, 14]. However, even when the problem can be expressed, these model-checkers do not provide much assurance that the semantics they use for proofs matches the compiled code. In the future, we would like to investigate integrating Pipit with a model-checker via an unverified extraction: such an extraction may allow some of the usability benefits such as counterexamples and invariant generation. If this integration were used solely for debugging and suggesting candidate invariants, then such a change would not necessarily expand the trusted computing base – that is, we could augment our end-to-end verified workflow with *unverified but validated* invariant generation.

Recent work has also introduced a form of refinement types for Lustre [12]. Rather than using transition systems, this work generates self-contained verification conditions based on the types of streams. Such a type-based approach promises to allow abstraction of the implementation details. However, for general-purpose functions such as *count_when* from Section 2, it is not clear how to give it a specification that actually *abstracts* the implementation: a simple specification that the result is within some range would hide too much and be insufficient for verifying the rest of the system. For such functions, the best specification is likely to include a re-statement of the implementation itself.

The embedded language Copilot generates real-time C code for runtime monitoring [28]. Recent work has used translation validation to show that the generated C code matches the high-level semantics [40]. Copilot supports model-checking via Kind2; however, the model-checking has a limited specification language and does not support contracts.

Early work embedding a denotational semantics of Lucid Synchrone in an interactive theorem prover focussed on the semantics itself, rather than proving programs [4]. There is ongoing work to construct a denotational semantics of Vélus for program verification [6]. We believe that the hybrid SMT approach of F^{*} will allow for a better mixture of automated

proofs with manual proofs. Compared to Vélus alone, the trusted computing base of Pipit is larger: we depend on all of F^{*}, Low^{*}'s unverified C code extraction and the Z3 SMT solver; in comparison, Vélus' C code generation is verified and does not depend on any SMT solver.

The deferred aspect of our proofs is similar to the deferred proofs of verification conditions for imperative programs, such as [32]. However, such verification conditions are *syntactically* deferred so that the verification condition can be proved later; in our case, the verification conditions are *semantically* deferred, so that more knowledge of the enclosing program can be exploited in the proof. In imperative programs, this sort of extra knowledge is generally provided explicitly as loop invariants, and non-looping statements have their weakest precondition computed automatically. In Lustre-style synchronous languages such as ours, programs tend to be composed of many nested recursive streams, which perform a similar function to loops. Explicitly specifying an invariant for each recursive stream would be cumbersome; deferring the proof allows such invariants to be implicit.

8 Conclusion

We have presented Pipit, a verified compiler and proof system for reactive systems. Our implementation of the TTCAN driver logic shows that, by embedding pure F^{*} functions for array operations, Pipit can express programs which are currently unsupported by other verified Lustre compilers. Pipit can also verify high-level program properties which are difficult to express and prove in existing Lustre model-checkers. Our development includes verified translations to both abstract and executable transition systems; both are shown to preserve the dynamic semantics. We also introduced a checked semantics, which describes the semantics of checked properties and contracts; proof obligations generated by translation to abstract transition system are verified to correspond to these semantics.

In the future, we intend to verify the remainder of the TTCAN driver logic. We also intend to increase the expressivity of Pipit by adding *clocks*, which are used to describe partially-defined streams [10]. Clocks are important for composing complex systems together and avoiding unnecessary computation; they may be useful if it becomes necessary to optimise the runtime of the TTCAN driver.

We are interested in further pursuing the intersection of model-checking with interactive theorem proving. A smart-contract called Djed [42] currently uses a mixture of Kind2 [11] and manual Isabelle/HOL proofs to show that the contract is well-behaved. In future work, we would like to further investigate whether Pipit's integration of streaming proofs with F^{*}'s automated proof system would be able to provide similar proofs, without introducing any semantic gap between the two systems.

References

- 1 ISO/CD 11898-4. Road vehicles - Controller area network (CAN) - Part 4: Time triggered communication. Standard, International Organization for Standardization, 2000.
- 2 Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016.
- 3 Dariusz Biernacki, Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. Clock-directed modular code generation for synchronous data-flow languages. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 121–130, 2008.
- 4 Sylvain Boulmé and Grégoire Hamon. A clocked denotational semantics for Lucid-Synchrone in Coq. *Rap. tech., LIP6*, 2001.

- 5 Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. A formally verified compiler for Lustre. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017.
- 6 Timothy Bourke, Paul Jeanmaire, and Marc Pouzet. Towards a denotational semantics of streams for a verified Lustre compiler, 2022. URL: https://types22.inria.fr/files/2022/06/TYPES_2022_slides_28.pdf.
- 7 Timothy Bourke, Basile Pesin, and Marc Pouzet. Verified compilation of synchronous dataflow with state machines. *ACM Transactions on Embedded Computing Systems*, 22(5s):1–26, 2023.
- 8 Aaron R Bradley. SAT-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation: 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23–25, 2011. Proceedings 12*. Springer, 2011.
- 9 Lélio Brun, Christophe Garion, Pierre-Loïc Garoche, and Xavier Thirioux. Equation-directed axiomatization of Lustre semantics to enable optimized code validation. *ACM Transactions on Embedded Computing Systems*, 22(5s):1–24, 2023.
- 10 Paul Caspi and Marc Pouzet. A functional extension to Lustre. *Intensional Programming I*, 1995.
- 11 Adrian Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli. The Kind 2 model checker. In *Computer Aided Verification*, 2016.
- 12 Jiawei Chen, José Luiz Vargas de Mendonça, Shayan Jalili, Bereket Ayele, Bereket Ngussie Bekele, Zhemin Qu, Pranjal Sharma, Tigist Shiferaw, Yicheng Zhang, and Jean-Baptiste Jeannin. Synchronous programming and refinement types in robotics: From verification to implementation. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Formal Techniques for Safety-Critical Systems*, 2022.
- 13 Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. Scade 6: A formal language for embedded critical software development. In *2017 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 1–11. IEEE, 2017.
- 14 Niklas Eén, Alan Mishchenko, and Robert Brayton. Efficient implementation of property directed reachability. In *2011 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2011.
- 15 Thomas Fuehrer, Bernd Mueller, Florian Hartwich, and Robert Hugel. Time triggered CAN (TTCAN). *SAE transactions*, pages 143–149, 2001.
- 16 Andrew Gacek, John Backes, Mike Whalen, Lucas Wagner, and Elaheh Ghassabani. The JKind model checker. In *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14–17, 2018, Proceedings, Part II 30*, pages 20–27. Springer, 2018.
- 17 Emilio Jesús Gallego Arias, Pierre Jouvelot, Sylvain Ribstein, and Dorian Desblancs. The W-calculus: a synchronous framework for the verified modelling of digital signal processing algorithms. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Functional Art, Music, Modelling, and Design*, pages 35–46, 2021.
- 18 Pranav Garg, Christof Löding, Parthasarathy Madhusudan, and Daniel Neider. ICE: A robust framework for learning invariants. In *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18–22, 2014. Proceedings 26*. Springer, 2014.
- 19 Léonard Gérard, Adrien Guatto, Cédric Pasteur, and Marc Pouzet. A modular memory optimization for synchronous data-flow languages: application to arrays in a Lustre compiler. *ACM SIGPLAN Notices*, 47(5), 2012.
- 20 Xiaoyun Guo, Toshiaki Aoki, and Hsin-Hung Lin. Model checking of in-vehicle networking systems with CAN and FlexRay. *Journal of Systems and Software*, 161:110461, 2020.
- 21 George Hagen and Cesare Tinelli. Scaling up the formal verification of Lustre programs with SMT-based techniques. In *2008 Formal Methods in Computer-Aided Design*. IEEE, 2008.
- 22 Florian Hartwich, Thomas Führer, Bernd Müller, and Robert Hugel. Integration of time triggered CAN (TTCAN_TC). *SAE Transactions*, pages 112–119, 2002.

- 23 Son Ho, Jonathan Protzenko, Abhishek Bichhawat, and Karthikeyan Bhargavan. Noise*: A library of verified high-performance secure channel protocol implementations. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 107–124. IEEE, 2022.
- 24 Erwan Jahier, Pascal Raymond, and Nicolas Halbwachs. The Lustre V6 reference manual. *Verimag, Grenoble, Dec*, 2016.
- 25 Kind2. Integer division rounds to negative infinite. Github issues, 2023. URL: <https://github.com/Kind2-mc/Kind2/issues/978>.
- 26 Kind2. *Kind2 user documentation*, 2.1.1 edition, 2023. URL: https://kind.cs.uiowa.edu/Kind2_user_doc/doc.pdf.
- 27 Kind2. Top-level array definition causes runtime failures. Github issues, 2024. URL: <https://github.com/Kind2-mc/Kind2/issues/1043>.
- 28 Jonathan Laurent, Alwyn Goodloe, and Lee Pike. Assuring the guardians. In *Runtime Verification: 6th International Conference, RV 2015, Vienna, Austria, September 22–25, 2015. Proceedings*. Springer, 2015.
- 29 Gabriel Leen and Donal Heffernan. Modeling and verification of a time-triggered networking protocol. In *International Conference on Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies (IC-NICONSIMCL'06)*, pages 178–178. IEEE, 2006.
- 30 Xin Li, Jian Guo, Yongxin Zhao, and Xiaoran Zhu. Formal modeling and verifying the TTCAN protocol from a probabilistic perspective. *Journal of Circuits, Systems and Computers*, 28(10):1950177, 2018.
- 31 Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Cătălin Hritcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, et al. Meta-F*: Proof automation with SMT, tactics, and metaprograms. 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*. Springer International Publishing Cham, 2019.
- 32 Liam O'Connor. Deferring the details and deriving programs. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development*, pages 27–39, 2019.
- 33 Can Pan, Jian Guo, Longfei Zhu, Jianqi Shi, Huibiao Zhu, and Xinyun Zhou. Modeling and verification of CAN bus with application layer using UPPAAL. *Electronic Notes in Theoretical Computer Science*, 309:31–49, 2014.
- 34 Jonathan Protzenko, Jean Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella Béguelin, Antoine Delignat-Lavaud, Catalin Hriteu, Karthikeyan Bhargavan, Cédric Fournet, et al. Verified low-level programming embedded in F*. *Proc. ACM program. lang.*, 1(ICFP), 2017.
- 35 Pascal Raymond. Synchronous program verification with Lustre/Lesar. *Modeling and Verification of Real-Time Systems*, 2008.
- 36 Robert Bosch GmbH. *M_TTCAN Time-triggered Controller Area Network User's Manual*, 3.3.0 edition, 2019. URL: https://www.bosch-semiconductors.com/media/ip_modules/pdf_2/m_can/mttcan_users_manual_v330.pdf.
- 37 Amos Robinson and Ben Lippmeier. Machine fusion: merging merges, more or less. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming*, pages 139–150, 2017.
- 38 Amos Robinson and Alex Potanin. Pipit: Reactive systems in F*(extended abstract). In *Proceedings of the 8th ACM SIGPLAN International Workshop on Type-Driven Development*, 2023.
- 39 Indranil Saha and Suman Roy. A finite state analysis of time-triggered CAN (TTCAN) protocol using Spin. In *2007 International Conference on Computing: Theory and Applications (ICCTA'07)*, pages 77–81. IEEE, 2007.

34:28 Pipit on the Post: Proving Pre- and Post-Conditions of Reactive Systems

- 40 Ryan G Scott, Mike Dodds, Ivan Perez, Alwyn E Goodloe, and Robert Dockins. Trustworthy runtime verification via bisimulation (experience report). *Proceedings of the ACM on Programming Languages*, 7(ICFP):305–321, 2023.
- 41 Michael Short and Michael J Pont. Fault-tolerant time-triggered communication using CAN. *IEEE transactions on Industrial Informatics*, 3(2):131–142, 2007.
- 42 Joachim Zahnentferner, Dmytro Kaidalov, Jean-Frédéric Etienne, and Javier Diaz. Djed: a formally verified crypto-backed autonomous stablecoin protocol. In *2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–9. IEEE, 2023.

Partial Redundancy Elimination in Two Iterative Data Flow Analyses

Reshma Roy¹ 

National Institute of Technology, Calicut, India

Sreekala S 

National Institute of Technology Calicut, India

Vineeth Paleri 

National Institute of Technology Calicut, India

Abstract

Partial Redundancy Elimination (PRE) is a powerful and well-known code optimization. The idea to combine Common Subexpression Elimination and Loop Invariant Code Motion optimizations into a single optimization was originally conceived by Morel and Renvoise. Their algorithm is bidirectional in nature and was not *complete* and *optimal*. Later, Knoop et al. proposed the first complete and optimal algorithm, Lazy Code Motion (LCM), which takes four unidirectional data flow analyses. In a recent paper, Roy et al. proposed an algorithm for PRE that uses three iterative data flow analyses. Here, we propose an efficient algorithm for PRE, which takes only two iterative data flow analyses followed by two computation passes over the program. The algorithm is both *computationally* and *lifetime* optimal. The proposed algorithm computes the information required for performing the transformation in two passes over the program without considering *safety*. The two iterative data flow analyses are required for making the transformation *safe*. The use of well-known data flow analyses, i.e., *available expressions* analysis and *anticipated expressions* analysis, makes the algorithm simple to understand and easy to prove its correctness. The proposed algorithm is more efficient than the existing algorithms since it takes only two iterative data flow analyses. The efficiency of the proposed algorithm is demonstrated by implementing it in LLVM Compiler Infrastructure and comparing the time taken with other selected best-known algorithms.

2012 ACM Subject Classification Software and its engineering → Compilers

Keywords and phrases Static Analysis, Data Flow Analysis, Code Optimization, Partial Redundancy Elimination

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.35

1 Introduction

Partial Redundancy Elimination (PRE) is a code optimization technique used in compiler design to eliminate redundant computations in a program. It focuses on identifying and eliminating computations that are partially redundant, i.e., the computations that occur more than once in a path in the input program. PRE helps reduce the number of instructions executed and can lead to significant performance improvements in a program. Partial redundancy elimination involves the insertion and deletion of computations at appropriate points in the program so that after the transformation, the program contains less than or equal number of occurrences of the original computation in any path. To preserve the semantics of the original program, the insertions of computations corresponding to the transformation must be safe, i.e., the program must not introduce new computations along any path in the original program.

¹ corresponding author

PRE can be either lexical-based or value-based. Lexical-based PRE focuses on eliminating lexically identical expressions on a path, while a value-based PRE eliminates expressions with identical values on a path. In this work, we focus on lexical-based partial redundancy elimination and anticipate that the insights from this work may find its application in the value-based approach as well. Here, we propose an efficient algorithm for partial redundancy elimination using two iterative data flow analyses followed by two computation passes over the program. The data flow analyses used are the well-known classical analyses, i.e., available expressions analysis and anticipated expressions analysis. Unlike the existing works [6, 9, 14, 16], the proposed algorithm requires only two iterative data flow analyses to perform partial redundancy elimination resulting in a significant efficiency gain. The contributions of our work are:

1. A new algorithm for lexical-based PRE, which takes only two iterative data flow analyses compared to at least three analyses in the existing well-known algorithms [6, 9, 14, 16].
2. Correctness proof of the proposed algorithm.
3. An experimental comparison of the proposed algorithm with the selected existing algorithms [9, 16] demonstrating its efficiency and precision.

1.1 Background

Morel and Renvoise, in their seminal work on partial redundancy elimination (PRE) [12], observed that an algorithm for partial redundancy elimination could potentially address both redundancy elimination and the loop invariant code motion simultaneously. Their approach involved four bidirectional data flow analyses. Morel and Renvoise's algorithm did not achieve computational optimality, i.e., it could not eliminate all partially redundant expressions in a program. Subsequently, Dhamdhere [5] improved upon Morel and Renvoise's algorithm by introducing the concept of edge placement, eliminating more partial redundancies. Another challenge in Morel and Renvoise's algorithm was the occurrence of redundant code motion, an issue that Dhamdhere [5] and Drechsler et al. [7] tackled as they implemented various improvements.

Knoop, Ruthing, and Steffen introduced the *lazy code motion algorithm* for partial redundancy elimination (PRE) [9], incorporating four unidirectional data flow analyses. This algorithm stands out for its computational and lifetime optimality, using a hoisting-followed-by-sinking approach. Knoop et al. devised a method to identify the *earliest* and *latest* points for performing the transformation. Another aspect of their algorithm is the preprocessing step, which involves inserting dummy nodes at the edges of nodes with multiple predecessors. Unfortunately, this step leads to unnecessary edge insertions, resulting in overhead. In response to these considerations, Knoop et al. later refined the lazy code motion algorithm to enhance its practical utility [10]. Additionally, Drechsler and Stadel [8] proposed a variant of the lazy code motion algorithm with a primary focus on practical applicability.

In the realm of partial redundancy elimination (PRE), Paleri et al. presented an algorithm utilizing classical data flow analyses, i.e., *availability*, *anticipability*, *partial availability*, and *partial anticipability* [14]. Notably, the introduction of the path concept in their paper enhances the algorithm's comprehensibility. Furthermore, this algorithm is both computationally and lifetime optimal. Originally designed for nodes containing single statements, Paleri et al. later modified their algorithm to nodes containing multiple instructions, such as the standard basic block [15]. In a work akin to the approach by Paleri et al., Dhamdhere introduced the concept of eliminability paths to address the optimal placement of computations [6]. Like those of prior researchers, Dhamdhere's approach relies on four unidirectional

analyses to eliminate partial redundancies. Recent work by Roy et al. [16] describes an algorithm for PRE that is more efficient than the other computationally optimal algorithms available in the literature since it takes only three iterative data flow analyses - *anticipated expressions*, *safe partially available expressions*, and *safe redundancy path* - compared to four analyses taken by the other algorithms.

One limitation of the presented PRE algorithm is its exclusive focus on lexically equivalent expressions. In contrast, a value-based PRE approach can potentially uncover a greater number of redundancies. The value-based method identifies equivalent expressions based on their actual values rather than relying solely on lexical equivalence. This distinction makes it a more powerful optimization technique for effectively eliminating redundant expressions, reported in the literature [4, 11, 13, 17].

2 Notations and Definitions

We found the formal definitions and notations from [14] appropriate for the proposed algorithm. In this section, we give an informal description of the terms used in the algorithm.

2.1 Control Flow graph

We represent a program as a Control Flow Graph (CFG) $G = (N, E, \text{entry}, \text{exit})$ where N represents the set of nodes in the graph and E is the set of edges in the graph. We assume that the CFG has two empty basic blocks, an *entry* node which represents the starting point of the graph and an *exit* node to which all exits of the graph go. An *entry* is the unique entry node with no predecessor nodes, and *exit* is the unique exit node without any successor nodes. Each node in the CFG contains at most one statement in the three-address code form. The assignment statement is of the form $x = e$, where x is a variable, and e is an expression built of variables, constants, and operators. The edge from node i to node j is represented as (i, j) . The sets of immediate predecessors and immediate successors of a node n are denoted as $\text{pred}(n)$ and $\text{succ}(n)$, respectively, where $\text{pred}(n) = \{m | (m, n) \in E\}$ and $\text{succ}(n) = \{m | (n, m) \in E\}$.

2.1.1 Annotated Control Flow Graph

An annotated control flow graph (ACFG) is a CFG annotated with the information obtained from a data flow analysis at every program point in the CFG, i.e., the input and output points of the basic blocks in the CFG.

2.2 Boolean Properties Associated with the Expressions

An expression e is said to be *locally available* from node i , i.e., available at the output point of node i (AvLoc_i), if e appears in node i , and the statement in node i does not modify the operands in e . An expression e is said to be *locally anticipated* from node i (ANTLoc_i), i.e., anticipated at the input point of node i if e appears in node i . An expression e is said to be *transparent* in node i (TRANSP_i), if the execution of the statement in node i does not modify the operands in e .

An expression is *available* at a point if it has been computed along all paths reaching this point with no changes to its operands since the computation. An expression is said to be *anticipated* at a point if every path from this point has a computation of that expression with no changes to its operands in between. An expression e is *partially available* at point p if there is at least one path from *entry* to p which computes e , and after the last such computation before reaching p there is no modification to its operands. An expression e , occurring at a point p , is partially redundant if e is *partially available* at p . An expression e is *partially anticipated* at p if there is at least one path from p to *exit* which computes e with no changes to its operands in between p and the point of occurrence of e . A point is *safe* for insertion of an expression e if the expression is either *available* or *anticipated* at that point. An expression is *safe partially available* at a point p if the expression is *partially available* at p and the path from point k to p is *safe*, where k is the point from which expression is *partially available* at p . The path formed by connecting adjacent program points where the expression is *safe partially available* is known as *safe partially available path*. An expression is said to be *safe partially anticipated* at a point p if the expression is *partially anticipated* at p and the path from point p to k is *safe*, where k is the point from which e is *partially anticipated* at p . A path is said to be a *safe redundancy path* for an expression if the expression is both *safe partially available* and *safe partially anticipated* at all points on the path.

The notations used for the properties defined in this section, corresponding to an expression e are described below:

Notations	Data flow properties
AvLOC_i	: Locally Available at the output point of node i
ANTLOC_i	: Locally Anticipated at the input point of node i
TRANSP_i	: Transparent in node i
$\text{AVIN}_i/\text{AVOUT}_i$: Available at input/output point of node i
$\text{ANTIN}_i/\text{ANTOUT}_i$: Anticipated at input/output point of node i
$\text{SPAVPATHIN}_i/\text{SPAVPATHOUT}_i$: Input/output point of node i is on Safe Partially Available Path
$\text{SREDPATHIN}_i/\text{SREDPATHOUT}_i$: Input/output point of node i is on Safe Redundancy Path

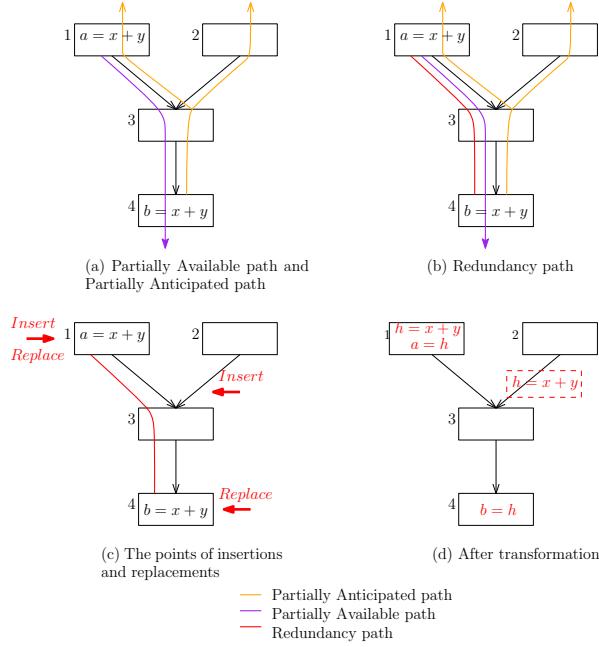
The properties for all the nodes in the CFG are expressed in terms of Boolean equations. We used the symbols summation/product (i.e., \sum / \prod) for the confluence operator, $+$ and \cdot for Boolean connectives *or* and *and*, and \neg for Boolean negation.

3 Basic Concept

We build on the basic concepts from [14]. The basic idea in [14] is briefly outlined below.

Partial Redundancy Elimination consists of two stages: detection and elimination. An expression e at a point p is said to be *partially redundant* if the expression is *partially available* at p . Thus, to detect partially redundant expressions, we require only the information regarding the *partially available expressions*. This information is obtained through *partially available expressions* analysis. In order to eliminate the partially redundant expressions, we require additional information on *partially anticipated expressions*, which is obtained through *partially anticipated expressions* analysis.

The fundamental idea behind partial redundancy elimination is to find *redundancy paths*. To identify the redundancy paths, we first *mark* all the program points where the expression under consideration is both *partially available* and *partially anticipated*. Now, we identify the redundancy paths by connecting the adjacent program points which are *marked*. Partial redundancy elimination is done by *insertions* and *replacements* of expressions



■ **Figure 1** An example for partial redundancy elimination.

at appropriate points in the program [See Fig. 1(c)]. As the initial transformation step, all partially redundant expressions are made totally redundant by inserting the statement $h = e$, where e is the expression of interest, at the edges that enter the junction nodes on the redundancy paths. Now, we insert the statement $h = e$ at the starting points of the redundancy paths. The next step involves the elimination of all the redundant expressions through replacements. The replacement involves the redundant expressions being replaced by the temporary variable h .

We consider the same example in [16] to explain the basic concept. In Fig. 1(a), the purple line represents the path where the expression is *partially available* at all points on the path. Similarly, the orange line denotes the path where the expression is *partially anticipated* at all points on the path. The *redundancy path* is marked in the red line in Fig. 1(b). The insertion points are the input point of node 1 and the edge (2, 3), and the replacement points are nodes 1 and 4, as shown in Fig. 1(c) based on the basic idea explained above. The CFG after the transformation is given in Fig. 1(d).

3.1 The New Approach

As stated above, *redundancy path* is the basic idea behind PRE. We observe an important characteristic of a *redundancy path* corresponding to an expression e . In a *redundancy path*, the first and last nodes contain the expression e [See Fig. 1(c)]. To identify this *redundancy path* for e , we need to visit the nodes in the CFG in a systematic fashion such that e must be *partially available* and *partially anticipated* in the nodes.

A *partially available path* for an expression e starts at a node (say s) containing e in the CFG and moves in the forward direction. We propagate the *partially available* information of e from the node s forward until e is killed or the exit node of the CFG is reached. A *partially anticipated path* for an expression e starts at a node (say t) containing e and moves

in the backward direction. We propagate *partially anticipated* information of e from the node t along the *partially available path* computed earlier until e is no longer *partially available*. The path from s to t along which the expression e is both *partially available* and *partially anticipated* forms the *redundancy path*. Thus, the *redundancy path* is obtained using just two computations – not two iterative data flow analyses.

To preserve the semantics of the original program, the insertions of computations corresponding to the transformation done during the PRE algorithm must be safe. We use the notion of *safety* to preserve the semantics of the transformed program. A point p is *safe* for insertion of an expression e if e is *available* or *anticipated* at p . Hence, instead of a simple *redundancy path*, we identify a *safe redundancy path* in the proposed algorithm. The information required to compute *safety* is obtained using two classical data flow analyses: *available expressions* analysis and *anticipated expressions* analysis. After computing *safety* information, *safe redundancy paths* are computed the same way as the computation of *redundancy paths* where propagation must additionally satisfy the *safety* property. Thus, the *safe redundancy path* is identified using just two computations: *safe partially available path* computation and *safe redundancy path* computation, which are detailed in Section 4.

Overall, the algorithm takes two iterative data flow analyses followed by two computation passes over the program.

4 The Proposed Algorithm for PRE

The proposed algorithm consists of two phases: a data flow analysis phase and a computation phase. The first phase has two classical unidirectional data flow analyses: *available expression* analysis and *anticipated expression* analysis. The second phase contains the computations for *safe partially available path* and *safe redundancy path*. The algorithm is presented for a single arbitrary expression e . However, an independent combination of all the expressions in a program will result in a global algorithm for partial redundancy elimination.

A detailed description of the data flow analyses and computations is presented in this section.

4.1 Data Flow Analysis Phase

4.1.1 Available Expression analysis

The available expression analysis (definition provided in Section 2.2) is done in the forward direction of the control flow graph. To solve the forward *available expression* analysis, we need to initialize $\text{AvOUT}_{\text{entry}}$ with the value FALSE because the expression is not available at the output point of the *entry* node. Note that an *entry* node is the first node of a CFG with no instructions in it. We initialize $\text{AvOUT}_i = \text{TOP}$ (TOP is denoted by \top) for all other nodes, as this value will allow the iterative algorithm to converge to the desired value. Note that for a value x , $x \wedge \top = x$. The iterative data flow analysis to compute available expression information is given in Algorithm 1.

Algorithm 1 Iterative data flow analysis to compute available expression information.

Input : Control Flow Graph(CFG), a program expression e .
Output: Input CFG annotated with *availability* information at all points for the expression e .

```

1 Procedure AvailExpr(CFG, e)
2   AvOUTentry = FALSE
3   for each node  $i \neq entry$  do
4     | AvOUTi =  $\top$ 
5   end
6   while changes to any AvOUT occur do
7     | for each node  $i \neq entry$  do
8       | | AvINi =  $\prod_{p \in pred(i)} AvOUT_p$ 
9       | | AvOUTi = AVLOCi + AvINi.TRANSPi
10      | end
11    end
12 end
```

4.1.2 Anticipated Expression analysis

The anticipated expression analysis (definition provided in Section 2.2) is carried out in the backward direction of the control flow graph. To solve the backward *anticipated expression* analysis, we need to initialize ANTIN_{exit} with the value FALSE because the expression is not anticipated at the input point of the *exit* node. We initialize ANTIN_i = \top for all other nodes, as this value will allow the iterative algorithm to converge to the desired value. The iterative data flow analysis to compute anticipated expression information is given in Algorithm 2.

Algorithm 2 Iterative data flow analysis to compute anticipated expression information.

Input : Control Flow Graph(CFG), a program expression e .
Output: Input CFG annotated with *anticipated* information at all points for the expression e .

```

1 Procedure AntExpr(CFG, e)
2   ANTINexit = FALSE
3   for each node  $i \neq exit$  do
4     | ANTINi =  $\top$ 
5   end
6   while changes to any ANTIN occur do
7     | for each node  $i \neq exit$  do
8       | | ANTOUTi =  $\prod_{s \in succ(i)} ANTIN_s$ 
9       | | ANTINi = ANTLOCi + ANTOUTi.TRANSPi
10      | end
11    end
12 end
```

4.2 Computation Phase

The second phase in the proposed algorithm consists of computations for *safe partially available path* and *safe redundancy path*. During this phase, the necessary information is computed by propagating data from specific points along predefined paths. It is important to note that the paths for data propagation are different for the two distinct computations. The worklist method is used to compute both computations.

4.2.1 Worklist

The basic idea of a work list is to maintain a list of nodes to be processed until the list becomes empty. There are three stages in the use of the worklist in the algorithm:

- **Initialization:** The WORKLIST is initialized with a set of nodes in the CFG containing the expression of interest.
- **Processing WorkList:**
 - GetNode: The node n to be processed next is taken out from the WORKLIST.
 - Process: Perform the computations on the node n .
 - Update: If there is a change in the value computed for the node n in the processing step, successor or predecessor nodes of n — for *safe partially available path* and *safe redundancy path* computations respectively — are added to the WORKLIST.
- **Termination:** The algorithm terminates when the WORKLIST becomes empty, indicating that all the required nodes are processed.

This worklist algorithm propagates the property, i.e., *safe partially available path* or *safe redundancy path*, from specific nodes, with which the worklist is initialized, through the nodes in the control flow graph until the property becomes FALSE. The algorithm is designed in such a way that each point in the CFG is processed only once.

The computations are detailed in the following sections.

4.2.2 Safe Partially Available Path Computation

In the initialization step of the *safe partially available path* computation, the nodes containing the expression of interest are collected and arranged in the reverse post-order sequence of their appearance within the CFG. This order facilitates efficient computation of information in the forward direction, commencing from each expression found within the CFG.

The basic idea is to compute *safe partially available path* for an expression by traversing a *safe path* in the forward direction and marking the points where the expression is also *partially available*. We get a *safe path* by connecting all the adjacent program points that are *safe*. Note that a point p is *safe* for insertion of an expression e if e is either *available* or *anticipated* at p .

The information required to compute *safety* is obtained during the first phase of the algorithm. After collecting *available* expression and *anticipated* expression information in the first phase, instead of computing *safety* as an independent computation, we integrate safety within the safe partially available path computation for efficiency. The computation of *safe partially available path* begins from a node with the expression of interest e and continues forward along the *safe path* until *partial availability* becomes FALSE. Note that *partial availability* becomes FALSE when expression e is *killed*.

Algorithm 3 Computation of safe partially available path for an expression e .

Input : Control Flow Graph annotated with *available* and *anticipated* information for e .

Output : Input CFG annotated with *safe partially available path* information for e .

```

1 Procedure SafeParAvailExpr(ACFG)
2   Create empty WORKLIST;
3   for each node  $i$  do // The order of traversal is reverse post order
4     SPAVPATHIN $_i$  = FALSE
5     SPAVPATHOUT $_i$  = FALSE
6     VISITEDIN $_i$  = FALSE
7     VISITEDOUT $_i$  = FALSE
8     if node  $i$  contains expression  $e$  then
9       WORKLIST.add( $i$ )
10    end
11   while !WORKLIST.isEmpty() do
12      $i$  = WORKLIST.remove()
13     if !VISITEDOUT $_i$  then
14       SPAVPATHOUT $_i$  = AvLOC $_i$  + SPAVPATHIN $_i$ . TRANSP $_i$  1
15       VISITEDOUT $_i$  = TRUE
16       if change to SPAVPATHOUT $_i$  occur then
17         for each node  $s \in \text{succ}(i)$  do
18           if !VISITEDIN $_s$  then
19             if SAFEIN $_s$  then // SAFEIN $_s$  = AvIN $_s$  + ANTIN $_s$ 
20               SPAVPATHIN $_s$  = TRUE
21               VISITEDIN $_s$  = TRUE
22               WORKLIST.add( $s$ )
23           end
24     end
25   end
```

4.2.3 Safe Redundancy Path Computation

In the initialization phase of the computation for *safe redundancy path*, the nodes containing the expression of interest are stored in the post-order sequence of their appearance within the CFG. This arrangement facilitates the efficient computation of information in a backward direction, commencing from each expression found within the CFG.

The basic idea is to compute *safe redundancy path* for an expression e by traversing a *safe partially available path* in the backward direction and marking the points where the expression is also *partially anticipated*. After computing *safe partially available path*, the *safe redundancy path* computation begins from a node in the initialized work list, and it progresses in a backward direction along the safe partially available path until the *partially anticipated* property becomes FALSE. Note that *partially anticipated* property becomes FALSE when expression e is *killed*.

¹ $\text{AvLOC}_i \implies \text{SAFEOUT}_i$ and $\text{SPAVPATHIN}_i \cdot \text{TRANSP}_i \implies \text{SAFEOUT}_i$

Algorithm 4 Computation of safe redundancy path for an expression e .

Input : Control Flow Graph annotated with *safe partially available path* information for e .

Output : Input CFG annotated with *safe redundancy path* information for e .

```

1 Procedure SafeRedPath(ACFG)
2   Create empty WORKLIST;
3   for each node  $i$  do           // The order of traversal is post order
4     SREDPATHIN $_i$  = FALSE
5     SREDPATHOUT $_i$  = FALSE
6     VISITEDIN $_i$  = FALSE
7     VISITEDOUT $_i$  = FALSE
8     if node  $i$  contains expression  $e$  then
9       WORKLIST.add( $i$ )
10    end
11   while !WORKLIST.isEmpty() do
12      $i$  = WORKLIST.remove()
13     if !VISITEDIN $_i$  then
14       SREDPATHIN $_i$  = SPAVIN $_i$ . (ANTLOC $_i$  + SREDPATHOUT $_i$ . TRANSP $_i$ )
15       VISITEDIN $_i$  = TRUE
16       if change to SREDPATHIN $_i$  occur then
17         for each node  $p \in \text{pred}(i)$  do
18           if !VISITEDOUT $_p$  then
19             if SPAVPATHOUT $_p$  then
20               SREDPATHOUT $_p$  = TRUE
21               VISITEDOUT $_p$  = TRUE
22               WORKLIST.add( $p$ )
23         end
24      end
25   end
26 
```

4.3 The Main Algorithm

The main algorithm for PRE is given in this section. After computing the required information from the two phases of the algorithm given in sections 4.1 and 4.2, the points of transformation are identified. We can divide the conceptual idea behind the algorithm into three stages:

1. Identification of partially redundant computations.
2. Conversion of partially redundant computations into totally redundant computations through insertions of expressions at program points identified. During insertions, we insert an assignment of the form $h = \text{expr}$, where h is a new temporary variable.
3. Elimination of all the redundant expressions through replacements at the identified program points. During replacements, we replace some of the original computations of expr by h .

We denote the insertion at the entry of node i by INSERT_i , insertion on edge (i, j) by $\text{INSERT}_{(i,j)}$, and replacement in node i by REPLACE_i . These terms compute Boolean values, and we use this information to detect the places of insertions and replacements. The proposed algorithm for partial redundancy elimination is given as Algorithm 5. The $\text{TRANSFORM}()$ function in Algorithm 5 does the necessary transformation using the information computed earlier in the algorithm.

Algorithm 5 Algorithm for Partial Redundancy Elimination.

Input : Control Flow Graph(CFG), a program expression e
Output: The input CFG with the partial redundancies of e eliminated.

```

1 Procedure PRE (CFG,  $e$ )
2   AVAILEXPR(CFG,  $e$ )
3   ANTEXPR(CFG,  $e$ )
4   SAFEPRAVAILPATH(ACFG)2           // A computation using work list
      algorithm
5   SAFEREDPATH(ACFG)3           // A computation using work list algorithm
6   for each node  $i$  in the CFG do
7     INSERT $_i$  =  $\neg$  SREDPATHIN $_i$  . SREDPATHOUT $_i$ 
8     REPLACE $_i$  = AVLOC $_i$ .SREDPATHOUT $_i$  + ANTLOC $_i$ .SREDPATHIN $_i$ 
9   end
10  for each edge  $(i,j)$  in CFG do
11    INSERT $_{(i,j)}$  =  $\neg$  SREDPATHOUT $_i$ 
12    .
13    SREDPATHIN $_j$ 
14  end
15  TRANSFORM(CFG, INSERT $_{1\dots n}$ , INSERT $_{(1\dots n, 1\dots n)}$ , REPLACE $_{1\dots n}$ )          /*  $n$ 
      represents the number of nodes in the CFG */
16 end
```

4.4 Example

In Fig. 2, we present an example [14] to illustrate the operation of the proposed algorithm. In Fig. 2(a), the blue line represents the anticipated path. In Fig. 2(b), the orange line shows the available path. The adjacent points which are either blue or orange are joined to form the safe path. The red dotted line in Fig. 2(b) represents safe path. The red line in Fig. 2(c) signifies the safe partially available paths. The brown line in Fig. 2(d) represents the safe redundancy path, which is computed by traversing the safe partially available path in a backward direction and identifying the points where the expression is also partially anticipated. The transformed CFG with insertions and replacements is shown in Fig. 3. The data flow analysis and the transformation information are given in Table 1.

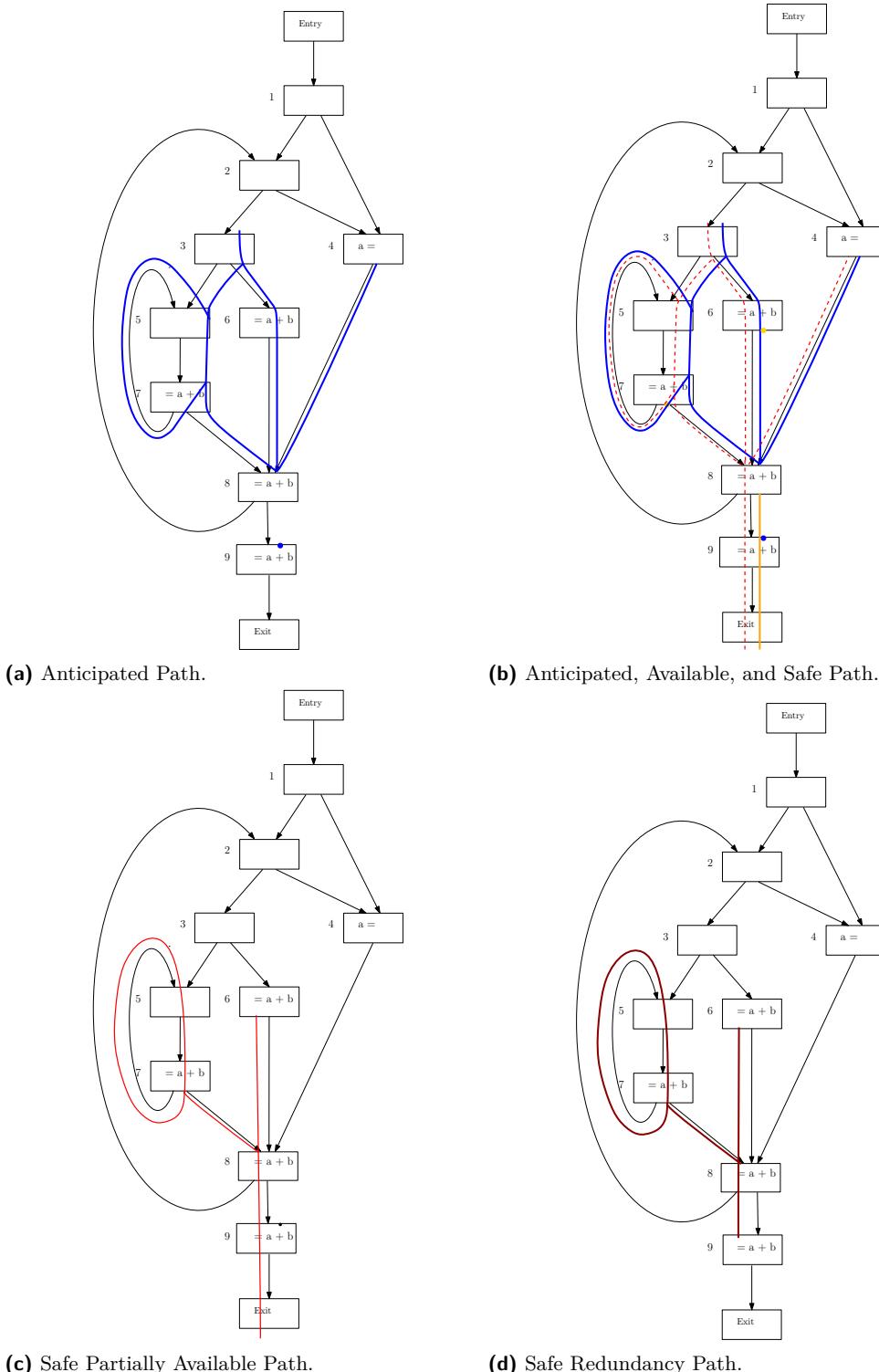
5 Proof of Correctness and Optimality

In this section, we prove the correctness of the analyses performed in the proposed PRE algorithm. In the algorithm, two well-known classical analyses are presented. Therefore, we only provide proof of the correctness of the algorithms for computations in the PRE algorithm. For the proof, as in the algorithm, we consider only one expression e in the input program. Also, our CFG nodes have only a single statement. We assume that a statement of the form $x = x + 1$ is transformed into two statements, $t = x + 1$ and $x = t$, where t is a unique temporary variable.

² ACFG with available and anticipated information for the expression e

³ ACFG with safe partially available path information for the expression e

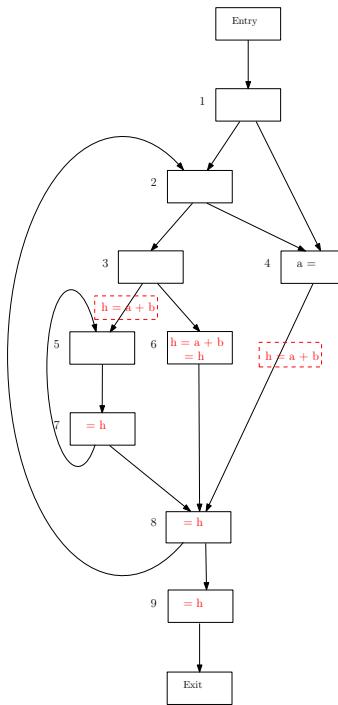
35:12 Partial Redundancy Elimination in Two Iterative Data Flow Analyses



■ **Figure 2** An Example demonstrating PRE by the proposed algorithm.

■ **Table 1** Boolean Properties and Transformations.

Local Boolean Properties	Global Boolean Properties	Insertions and Replacements
$\text{AvLOC}_i = \{6, 7, 8, 9\}$	$\text{ANTIN}_i = \{3, 5, 6, 7, 8, 9\}$	$\text{INSERT}_i = \{6\}$
$\text{ANTLOC}_i = \{6, 7, 8, 9\}$	$\text{ANTOUT}_i = \{3, 4, 5, 6, 7\}$	$\text{INSERT}_{(i,j)} = \{(3, 5), (4, 8)\}$
	$\text{AVIN}_i = \{9\}$	$\text{REPLACE}_i = \{6, 7, 8, 9\}$
	$\text{AVOUT}_i = \{6, 7, 8, 9\}$	
	$\text{SAFEIN}_i = \{3, 5, 6, 7, 8, 9\}$	
	$\text{SAFEOUT}_i = \{3, 4, 5, 6, 7, 8, 9\}$	
	$\text{SPAVPATHIN}_i = \{5, 7, 8, 9\}$	
	$\text{SPAVPATHOUT}_i = \{5, 6, 7, 8, 9\}$	
	$\text{SREDPATHIN}_i = \{5, 7, 8, 9\}$	
	$\text{SREDPATHOUT}_i = \{5, 6, 7, 8\}$	



■ **Figure 3** CFG after transformation.

5.1 Correctness of Safe Partially Available Path computation

► **Theorem 1** (Correctness). *The computation of the safe partially available path is done correctly.*

Proof. We have to show that every point computed as safe partially available by the safe partially available path computation (Algo.3) is correct. Let N represent the set of nodes in the input CFG.

Axiom 1. $\{\forall i: i \in N: (\text{SPAVPATHIN}_i = \text{FALSE}) \wedge (\text{SPAVPATHOUT}_i = \text{FALSE})\}$ at the beginning.

[From initialisation in lines 4-5]

Axiom 2. For an expression e , the input point of node i is on a safe partially available path if the input point of i is safe and the output point of at least one predecessor of node i is on the safe partially available path.

i.e., $\text{SPAVENT}_{\text{IN}_i} = \text{SAFEIN}_i \cdot (\sum_{p \in \text{pred}(i)} \text{SPAVENT}_{\text{OUT}_p})$ [By definition, Section 2.2]

Axiom 3. In the algorithm, $\text{SPAVENT}_{\text{IN}_i}$ is set to TRUE for a node i iff
 $\{\exists p: p \in \text{pred}(i): \text{SPAVENT}_{\text{OUT}_p} \cdot \text{SAFEIN}_p\}$ [Lines 16, 19-20]

► **Lemma 2.** *The computation of the safe partially available path at the input point of node i , i.e., $\text{SPAVENT}_{\text{IN}_i}$, is done correctly.*

Proof. Proof is as follows:

$$\begin{array}{ll} \text{Axiom 2 and Axiom 3} & \Rightarrow \text{SPAVENT}_{\text{IN}_i} \text{ is set to TRUE at the input point of node } i \\ & \text{if and only if safe partial availability is true} \quad - (1) \\ (1) \text{ and Axiom 1} & \Rightarrow \text{The input point of node } i \text{ which is not safe partially available remains FALSE} \\ & \quad - (2) \\ (1) \text{ and (2)} & \Rightarrow \text{Lemma 2} \end{array} \blacktriangleleft$$

Axiom 4. For an expression e , the output point of node i is on a safe partially available path, if e is locally available or the input point of node i is on safe partially available path and e is transparent in i . i.e., $\text{SPAVENT}_{\text{OUT}_i} = \text{AVLOC}_i + \text{SPAVENT}_{\text{IN}_i} \cdot \text{TRANSP}_i$. [By definition, Section 2.2]

► **Lemma 3.** *The computation of the safe partially available path at the output point of node i , i.e., $\text{SPAVENT}_{\text{OUT}_i}$, is done correctly.*

Proof. $\text{SPAVENT}_{\text{OUT}_i}$ is changed only for the nodes that are added to the work list. Therefore, we consider the nodes that are added to the work list. If a node i is added to the work list, then either of the following cases holds.

Case 1. Node i contains expression e . [Line 9]

$$\begin{aligned} &\Rightarrow \text{AVLOC}_i && - (3) \\ &\text{[Note that in our CFG, a block has only one instruction. Also, an instruction of the form } x = x + 1 \text{ is transformed into two statements,} \\ &\quad t = x + 1 \text{ and } x = t.] \\ &\Rightarrow \text{SPAVENT}_{\text{OUT}_i} && \text{[By Axiom 4]} - (4) \end{aligned}$$

Case 2. Node i does not contain the expression e (i.e. AVLOC_i is FALSE) and $\text{SPAVENT}_{\text{IN}_i}$ is TRUE. [Lines 20, 22] – (5)

We need to prove that, under the condition (5), $\text{SPAVENT}_{\text{OUT}_i}$ is set to TRUE if and only if TRANSP_i is TRUE (as given in line 14).

$$\begin{aligned} \text{SPAVENT}_{\text{OUT}_i} &\equiv \text{AVLOC}_i + \text{SPAVENT}_{\text{IN}_i} \cdot \text{TRANSP}_i && \text{[By Axiom 4]} \\ &\equiv \text{FALSE} + \text{SPAVENT}_{\text{IN}_i} \cdot \text{TRANSP}_i && \text{[AVLOC}_i = \text{FALSE, From (5)]} \\ &\equiv \text{SPAVENT}_{\text{IN}_i} \cdot \text{TRANSP}_i && \text{[FALSE} + \text{P} \equiv \text{P}] \\ &\equiv \text{TRUE} \cdot \text{TRANSP}_i && \text{[SPAVENT}_{\text{IN}_i} = \text{TRUE, From (5)]} \\ &\equiv \text{TRANSP}_i && \text{[TRUE} \cdot \text{P} \equiv \text{P}] \end{aligned}$$

i.e., $\text{SPAVENT}_{\text{OUT}_i}$ is TRUE iff node i is transparent.

Hence, $\text{SPAVENT}_{\text{OUT}_i}$ is set to TRUE correctly in case 2. – (6)

- (4) and (6) \Rightarrow SPAVPATHOUT_i is set to TRUE at the output point of node *i* if and only if safe partial availability is true. – (7)
- (7) and Axiom 1 \Rightarrow The output point of node *i* which is not safe partially available remains FALSE. – (8)
- (7) and (8) \Rightarrow Lemma 3

Lemma 2 and Lemma 3 \Rightarrow Theorem 1

► **Theorem 4** (Completeness). *The computation of the safe partially available path identifies all points that are safe partially available.*

Proof. We take three stages in the computation to prove the completeness:

- *Starting node of a safe partially available path:* A safe partially available path begins at the output point of a node containing the expression *e*. A node *i* containing the expression *e* is added to the work list in lines 8-9. The node *i* is then taken out from the work list (line 12) and safe partial availability information at the output point of node *i* is computed correctly in line 14.
- *Propagation of information:* The safe partial availability information at the output point of node *i* is then propagated to the input point of each of the successor nodes, say *j*, if the input point of node *j* is safe (lines 17-20), and those successor nodes are added to the work list (line 22). Each of these successor nodes is later taken out from the work list (line 12), and the information is further propagated from the input point to the output point of node *j* if node *j* is transparent (line 14).
- *End node of a safe partially available path:* The propagation ends under two conditions:
 - (i) The propagation from the input point to the output point of node *i* ends if *e* is killed in *i*.
 - (ii) The propagation from the output point of node *i* to the input point of its successor node *j* ends if input point of node *j* is not safe.

Hence, all possible safe partially available paths starting from a node *i* are computed correctly during this process.

This process of propagation of safe partial availability information is performed from each node containing *e* in the given input program. Hence, the computation identifies all points on the safe partially available path.

► **Theorem 5** (Termination). *Safe partially available path computation terminates.*

Proof. The algorithm terminates when the work list is empty (line 11). Initially, the work list contains nodes with the expression *e* from the input program (lines 8-9). After that, a node *i* is added to the work list if there is a change of value in SPAVPATHOUT_p where *p* \in pred(*i*) (lines 16, 22). The value in SPAVPATHOUT_i of a node *i* can change from the initialized value FALSE (line 5) to TRUE at most once (line 14), owing to the fact that once the value becomes TRUE, it remains TRUE. Hence, the number of nodes added to the work list after initialization equals the number of value changes for SPAVPATHOUT. If the total number of nodes in the CFG is *N*, then there can be at most *N* number of value changes. Since the nodes from the work list are removed (line 12) for computing SPAVPATHOUT, and the number of node additions is at most *N*, eventually the work list becomes empty. Hence, the algorithm terminates.

5.2 Correctness of Safe Redundancy Path computation

► **Theorem 6** (Correctness and Completeness). *The computation of the safe redundancy path is correct and complete.*

The line of reasoning is similar to the reasoning given for *safe partially available path*, except for the fact that the propagation in this case is in the backward direction and necessary changes accordingly. Hence, the formal proof is avoided here.

5.3 Optimality of Transformation

► **Theorem 7.** *The transformation in the proposed PRE algorithm is computationally and lifetime optimal.*

The proposed algorithm is based on the idea of the safe redundancy path in [14]. The transformation done on the safe redundancy path is proved to be both computationally and lifetime optimal in [14].

6 Experimental Results

In this section, we perform an experimental evaluation to compare the proposed algorithm with existing ones. For comparison, we consider two aspects: the number of redundancies detected (i.e., precision) and the running time of the algorithms. We have selected two of the existing algorithms which are computationally and lifetime optimal for comparison. The algorithms chosen for this comparison are: LCM [9], the well-known PRE algorithm which takes four analyses, and PRE-3 [16], which takes three analyses.

In the proposed work, the algorithm is designed for an arbitrary expression e . For implementation, we employ bit-vector representation to extend the algorithm to all the n expressions within the program. At a program point in the CFG, each property (e.g., SPAVPATHOUT_i in Algo. 3) is represented by a bit vector. Each bit in the bit vector corresponds to an expression where TRUE means the property is true for the expression, while FALSE means the property is false.

To illustrate how a bit vector facilitates parallel computation of all n expressions within the program, let's examine the computation $\text{SPAVPATHOUT}_i = \text{AvLOC}_i + \text{SPAVPATHIN}_i \cdot \text{TRANSP}_i$ in Algorithm 3. Consider the computation $\text{SPAVPATHIN}_i \cdot \text{TRANSP}_i$, where SPAVPATHIN_i and TRANSP_i are bit-vectors representing the information for n expressions. An AND operation between the bit-vectors SPAVPATHIN_i and TRANSP_i results in the bit-vector representing the property SPAVPATHOUT_i for all the n expressions at the output point of node i .

We used LLVM compiler infrastructure [1, 2] for our implementation. The results were obtained on a machine with a 1.8 GHz Intel Core i5 processor having 8 GB RAM for selected programs from the SPEC CPU2006 benchmark suite [3]. The analyses are intraprocedural. The algorithm is implemented for demonstrating its *completeness* and *efficiency*. Accordingly, we have decided to consider a subset of instructions i.e., instructions involving signed and unsigned integer arithmetic operators (+, -, *, ÷, %) to simplify the implementations. The LLVM IR instructions considered are *add*, *sub*, *mul*, *udiv*, *sdiv*, *urem* and *srem* as well as the *load* and *store* instructions of normal variables which includes both local and global variables. For other instructions, we made conservative assumptions. For example, consider a statement with pointer assignment, $*p = \dots$. This statement may change the value of normal variables of the program. So, we made a conservative assumption that all the variables are killed at the output point of such an instruction.

For our experiment, we begin with some preprocessing steps. We employ the *-instnamer* pass in LLVM to assign names to any unnamed values within the LLVM IR code. This is necessary as these values are not accessible through the *getName()* method we have used. We wanted only the instructions that can be reached from the entry node. To achieve this, we execute the *-unreachableblockelim* pass provided by LLVM. For Algorithm 3 and Algorithm 4, the worklist is implemented with the *InstructionWorkList* in llvm. This *InstructionWorkList* is implemented using a stack in llvm.

6.1 Efficiency

In this section, we compare the execution time of the proposed algorithm against the other two chosen algorithms: the LCM algorithm developed by Knoop et al. and PRE-3 by Roy et al. The algorithms were implemented as passes in the LLVM compiler and were run on selected programs from the SPEC CPU2006 benchmark suite [3] using the *-time-pass* optimizer tool of LLVM to measure execution time. The time taken for analyses by the CPU is measured where the reported time is the sum of the CPU time in user mode and the CPU time in system mode. We execute each benchmark program ten times, employing the *time-pass* functionality. We then calculate the average time from these ten runs. The time taken for analysis by each algorithm is then presented in seconds.

In Table 2, the second column displays the overall count of LLVM IR instructions within each benchmark program. The third column provides the total count of expressions considered, adhering to our conservative assumptions. The subsequent columns provide the time taken by each algorithm under consideration. The final row of the table presents the average time taken by each algorithm, taking into account all the selected benchmark programs.

The proposed algorithm performs better since it takes only two iterative data flow analyses compared to four by LCM and three by PRE-3. The proposed algorithm achieves 51% and 21% reduction in time over LCM and PRE-3, respectively, for the selected set of benchmark programs. The experimental results demonstrate that the proposed algorithm is more efficient in terms of the time taken for analysis compared to the other algorithms.

6.2 Precision

This section looks at the precision of the chosen algorithms, specifically focusing on their completeness in identifying redundant expressions. For the LCM algorithm, we record the count of insertions identified at nodes, the count of insertions specifically at the dummy nodes generated during preprocessing, replacement counts, and the total number of redundant expressions identified. In the case of PRE-3 and the proposed algorithm, we present the count of node insertions, edge insertions, replacements, and the total number of redundant expressions identified. Table 3 provides the information computed during the process. The total number of node insertions for LCM is displayed in column 2, which includes dummy nodes. Column 3 displays the number of dummy nodes created by the algorithm for LCM and used for insertions. Dummy node insertions in LCM are the *edge insertions* in PRE-3 and the Proposed Algorithm. The table demonstrates that the proposed algorithm detects the same number of redundancies as LCM and PRE-3, affirming the completeness of the algorithm. Moreover, upon examining the data in the table, it becomes evident that the identified points of insertions, replacements, and edge insertions are the same for all three algorithms.

Table 2 Comparison of Efficiency.

Benchmark Programs	No.of instructions	Expressions considered	Time (Seconds)		
			LCM	PRE-3	Proposed Algorithm
astar	11887	260	1.06	0.69	0.63
bzip2	27346	694	33.99	24.87	21.62
gcc	339578	1511	450.52	222.95	170.30
gromacs	185285	3605	40.56	27.71	23.96
h264ref	188827	6302	285.44	218.84	161.06
hmmer	90070	2077	19.47	13.22	11.60
lbbm	6155	1131	0.24	0.21	0.14
mcf	3917	90	0.24	0.18	0.16
povray	232142	2049	54.67	36.73	31.87
sjeng	32215	1460	22.60	13.51	12.79
soplex	133448	996	12.81	10.13	9.68
sphinx	47367	824	6.47	4.68	4.13
Average running time			77.33	47.81	37.32

Table 3 Comparison of Precision.

Benchmark Programs	LCM				PRE-3				Proposed Algorithm			
	Insertions (nodes in original CFG + dummy nodes added)	Insertions (dummy nodes)	Replace ments	Redundant expressions detected	Insertions (node)	Insertions (edge)	Replace ments	Redundant expressions detected	Insertions (node)	Insertions (edge)	Replace ments	Redundant expressions detected
astar	13	3	34	24	10	3	34	24	10	3	34	24
bzip2	39	3	82	46	36	3	82	46	36	3	82	46
gcc	61	16	135	90	45	16	135	90	45	16	135	90
gromacs	262	98	532	368	164	98	532	368	164	98	532	368
h264ref	686	99	2057	1470	587	99	2057	1470	587	99	2057	1470
hmmer	220	51	580	411	169	51	580	411	169	51	580	411
lbbm	132	0	919	787	132	0	919	787	132	0	919	787
mcf	10	1	45	36	9	1	45	36	9	1	45	36
povray	136	32	317	213	104	32	317	213	104	32	317	213
sjeng	161	13	363	215	148	13	363	215	148	13	363	215
soplex	48	14	70	36	34	14	70	36	34	14	70	36
sphinx	38	10	66	38	28	10	66	38	28	10	66	38

7 Conclusion

In this paper, we presented a novel algorithm for lexical-based partial redundancy elimination. The proposed algorithm takes two iterative data flow analyses followed by two computation passes over the program to perform the transformation. The use of well-known data flow analyses, i.e., *available expressions* analysis and *anticipated expressions* analysis, makes it easy to comprehend the algorithm and prove its correctness. We have provided the proof for the correctness of the algorithm. The algorithm is more efficient compared to other computationally and lifetime optimal algorithms in the literature, as it takes only two iterative data flow analyses, in contrast to at least three analyses required by other methods. The algorithm is both computationally and lifetime optimal. To substantiate these claims, we implemented the algorithm using the LLVM Compiler Infrastructure and compared the number of redundant expressions detected and the time taken for analyses against the existing algorithms. The results from the experiments conducted demonstrate

that the proposed algorithm detects the same number of redundant expressions and performs significantly better compared to the existing well-known algorithms considered. Although our algorithm is lexical-based, we believe that its fundamental principles hold significant potential for guiding the transition to a value-based approach, which could ultimately result in an efficient value-based PRE.

References

- 1 The LLVM Compiler Infrastructure Project. <http://llvm.org/>. Accessed on 03/07/2021.
- 2 LLVM programmer's manual. <https://llvm.org/docs/ProgrammersManual.html>. Accessed on 20-08-2021.
- 3 The SPEC CPU2006 benchmark suit. <https://www.spec.org/cpu2006/>, 2006. Accessed on 10-01-2022.
- 4 Rastislav Bodík and Sadun Anik. Path-sensitive value-flow analysis. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 237–251, New York, NY, USA, 1998. Association for Computing Machinery. doi:10.1145/268946.268966.
- 5 D. M. Dhamdhere. A fast algorithm for code movement optimisation. *ACM SIGPLAN Not.*, 23(10):172–180, October 1988. doi:10.1145/51607.51621.
- 6 Dhananjay M. Dhamdhere. E_path-PRE: Partial redundancy elimination made easy. *ACM SIGPLAN Not.*, 37(8):53–65, August 2002. doi:10.1145/596992.597004.
- 7 KarlHeinz Drechsler and Manfred P. Stadel. A solution to a problem with Morel and Renvoise's global optimization by suppression of partial redundancies. *ACM Trans. Program. Lang. Syst.*, 10(4):635–640, October 1988. doi:10.1145/48022.214509.
- 8 Karl-Heinz Drechsler and Manfred P. Stadel. A variation of Knoop, Rüthing, and Steffen's lazy code motion. *ACM SIGPLAN Not.*, 28(5):29–38, May 1993. doi:10.1145/152819.152823.
- 9 Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. *ACM SIGPLAN Not.*, 27(7):224–234, July 1992. doi:10.1145/143103.143136.
- 10 Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: Theory and practice. *ACM Trans. Program. Lang. Syst.*, 16(4):1117–1155, July 1994. doi:10.1145/183432.183443.
- 11 Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Expansion-based removal of semantic partial redundancies. In Stefan Jähnichen, editor, *Compiler Construction*, pages 91–106, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.
- 12 E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Commun. ACM*, 22(2):96–103, February 1979. doi:10.1145/359060.359069.
- 13 Rei Odaira and Kei Hiraki. Partial value number redundancy elimination. In Rudolf Eigenmann, Zhiyuan Li, and Samuel P. Midkiff, editors, *Languages and Compilers for High Performance Computing*, pages 409–423, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
- 14 Vineeth Kumar Paleri, Y. N. Srikant, and Priti Shankar. A simple algorithm for partial redundancy elimination. *ACM SIGPLAN Not.*, 33(12):35–43, December 1998. doi:10.1145/307824.307851.
- 15 Vineeth Kumar Paleri, Y. N. Srikant, and Priti Shankar. Partial redundancy elimination: a simple, pragmatic, and provably correct algorithm. *Sci. Comput. Program.*, 48(1):1–20, 2003. doi:10.1016/S0167-6423(02)00083-7.
- 16 Reshma Roy and Vineeth Paleri. Lexical-based partial redundancy elimination: An optimal algorithm with improved efficiency. *Journal of Computer Languages*, 75:101204, 2023. doi:10.1016/j.cola.2023.101204.
- 17 Thomas VanDrunen and Antony L. Hosking. Value-based partial redundancy elimination. In Evelyn Duesterwald, editor, *Compiler Construction*, pages 167–184, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

Scaling Interprocedural Static Data-Flow Analysis to Large C/C++ Applications

An Experience Report

Fabian Schiebel 

Fraunhofer Institute for Mechatronic Systems Design IEM, Paderborn, Germany

Florian Sattler 

Saarland University, Saarland Informatics Campus, Saarbrücken, Germany

Philipp Dominik Schubert 

Heinz Nixdorf Institute, Paderborn, Germany

Sven Apel 

Saarland University, Saarland Informatics Campus, Saarbrücken, Germany

Eric Bodden 

Paderborn University, Department of Computer Science, Heinz Nixdorf Institute, Germany

Fraunhofer IEM, Paderborn, Germany

Abstract

Interprocedural data-flow analysis is important for computing precise information on whole programs. In theory, the popular algorithmic framework *interprocedural distributive environments* (IDE) provides a tool to solve distributive interprocedural data-flow problems efficiently. Yet, unfortunately, available state-of-the-art implementations of the IDE framework start to run into scalability issues for programs with several thousands of lines of code, depending on the static analysis domain. Since the IDE framework is a basic building block for many static program analyses, this presents a serious limitation. In this paper, we report on our experience with making the IDE algorithm scale to C/C++ applications with up to 500 000 lines of code. We analyze the IDE algorithm and its state-of-the-art implementations to identify their weaknesses related to scalability at both a conceptual *and* implementation level. Based on this analysis, we propose several optimizations to overcome these weaknesses, aiming at a sweet spot between reducing running time and memory consumption. As a result, we provide an improved IDE solver that implements our optimizations within the PhASAR static analysis framework. Our evaluation on real-world C/C++ applications shows that applying the optimizations speeds up the analysis on average by up to 7×, while also reducing memory consumption by 7× on average as well. For the first time, these optimizations allow us to analyze programs with several hundreds of thousands of lines of LLVM-IR code in reasonable time and space.

2012 ACM Subject Classification Theory of computation → Program analysis

Keywords and phrases Interprocedural data-flow analysis, IDE, LLVM, C/C++

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.36

Supplementary Material Software (*Source Code + Experiment Results*): <https://doi.org/10.5281/zenodo.13137082>

Funding This work was partially supported by the Fraunhofer Internal Programs under Grant No. PREPARE 840 231, and by the German Research Foundation under Grant No. AP 206/11-2, and within the Collaborative Research Center TRR 248 under Grant No. 389792660.

1 Introduction

Over the recent years static program analysis has become an important tool for finding bugs and security vulnerabilities [7, 11, 16, 26–28, 30]. To produce results that actually help developers in these tasks, static analyses are ideally both sound (or at least soundy [14]) and precise, i.e., they report only true findings without missing any real bugs and vulnerabilities. The analyses need to obtain a complete picture about the program under analysis and therefore have to be interprocedural, i.e., following procedure calls. But it is a major challenge to develop sound and precise inter-procedural analyses that scale well with large real-world target programs [6, 19, 31, 32].

The *interprocedural distributive environments* (IDE) framework [20] operates on data-flow problems whose flow functions distribute over the analysis' merge operator. Following the functional approach to interprocedural analysis [24], for such distributive data-flow problems IDE constructs fine-grained, per-fact, procedure summaries that can be reapplied in each subsequent calling context of a given procedure. This allows IDE to scale to larger programs relatively well even though its time complexity is $\mathcal{O}(|N| \cdot |D|^3)$, where N is the set of nodes of the target program's interprocedural control-flow graph and D is the symbol domain of the data-flow analysis.

Common static analysis frameworks such as Heros [5] and PhASAR [22] provide generic and parameterizable IDE solver implementations; they even implement the simpler IFDS [17] algorithm in terms of IDE. For an analysis problem on the desired target program to be solved in an automated manner, users of these frameworks merely have to specify its flow (and edge) functions and provide this specification to the IDE implementation. Current IDE implementations, also known as solvers, aim at analyzing real-world target programs in a fully flow and context-sensitive manner, computing precise and informative results depending on the quality of the flow (and edge) functions' specification. Nonetheless, the authors of this paper can tell from many years of experience in program analysis that all publicly available IDE implementations run into severe scalability issues for larger target programs – a major problem. This effectively impedes or even prevents the analysis of many real-world programs, or forces analysis developers to resort to simpler analysis domains, which reduces the precision and usefulness of the analysis results. Sattler et al., for instance, present a novel concept to combine program analysis and repository mining that addresses numerous relevant software engineering problems [21]. This approach, however, requires one to run an exhaustive IDE-based taint analysis that needs to generate and propagate *all* program variables, which, in turn, produces millions of data flows. In this vein, we use PhASAR's current IDE implementation to demonstrate that sound and precise analyses that produce more than 100 million data flow edges cannot be completed using ordinary consumer hardware. Such a huge number of data flows can easily arise already when analyzing programs that comprise fewer than 100 000 instructions in LLVM's [13] intermediate representation (IR). The number of IR instructions is relevant, since PhASAR performs its analyses on the LLVM-IR level, and even seemingly small C/C++ programs can lead to a large number of IR instructions. Still, using an IR enables analysis writers to develop analyses for programs originating from complex languages, such as C++, that would otherwise add drastic implementation overhead. Further, we can support analyzing programs from multiple different source languages (in our case C and C++) with just one analysis implementation, whereas a source-level analysis would need different implementations per language. Therefore, we prefer analyzing LLVM IR and handle the program size from within the solver.

In this work, we report on our experiences analyzing real-world programs with the IDE framework, identifying two critical optimization levers when implementing a generic state-of-the-art IDE solver. Specifically, using 31 real-world C and C++ target programs, we evaluate PhASAR’s state-of-the-art IDE solver implementation with regard to runtime and memory consumption. Based on insights gained from these experiments, we propose and evaluate two optimizations that we have devised to improve the performance of the IDE implementation. One optimization chooses an optimized data layout for storing required data, while the other one extends the garbage collection procedure from Arzt [1].

The improved IDE solver, which incorporates the abovementioned optimizations and insights, reduces analysis running times as well as memory consumption by up to $7\times$ on average, depending on the client-analysis problem that should be solved. The experiments show that this allows one to conduct sound and precise inter-procedural data-flow analyses on interesting target programs such as FASTDOWNWARD, a domain-independent planning system, in reasonable time and space.

In summary, we make the following contributions:

- We analyze the IDE algorithm as described in the literature and its state-of-the-art, openly-available implementations with regard to runtime and memory consumption.
- Based on the analysis, we propose optimizations that overcome these weaknesses.
- We report on an empirical study on our optimized IDE solver, showing that it improves runtime and memory usage of IDE-based analysis by up to $7\times$ on average.
- We provide an open-source implementation of the IDE algorithm that incorporates our optimizations within PhASAR [22] and make it available as supplementary material¹.

The remainder of this paper is structured as follows: Section 2 gives an introduction to the IDE algorithm and Section 3 analyzes the state-of-the-art in IDE-based analysis and describes the problems that we identify. Section 4 presents our optimizations to IDE to mitigate these problems and Section 5 describes the highlights of our implementation. In Section 6, we detail on our empirical evaluation on real-world C/C++ programs and Section 8 concludes this paper.

2 Background on IDE

In this section, we introduce the conceptual *Interprocedural Distributive Environments* (IDE) [20] algorithm. IDE solves a data-flow problem by constructing an *exploded supergraph* (ESG). By construction, a data-flow fact d holds at instruction n , if a node (n, d) in the ESG is reachable from a special, tautological node (n_0, Λ) for an entry point statement n_0 . The ESG is constructed by replacing each node in the target program’s *interprocedural control-flow graph* (ICFG) with a bipartite graph representation of the respective flow functions. IDE requires all flow-functions to distribute over the merge operator (usually set union). Such distributive flow functions can be represented as bipartite graphs without loss of precision. The common flow functions *identity*, *gen* (generate), and *kill* (remove) are distributive and thus, all *gen/kill* data-flow problems can be encoded in IDE.

To enable a context-sensitive, interprocedural analysis, IDE follows the summary-based approach [24] to inter-procedural static data-flow analysis: It constructs per-fact summaries for sequences of instructions by composing their flow functions. The composition $h = g \circ f$ of two flow functions f and g , called *jump function*, can be produced by merging the nodes

¹ Supplementary Material: <https://zenodo.org/doi/10.5281/zenodo.13137081>

of g with the corresponding nodes of the domain of f . A jump function ranging from a given procedure p 's starting point to its exit point builds up a summary ψ of p . Once summary ψ has been constructed for procedure p , it can be re-applied in any other context in which the procedure p is called. The runtime complexity of IDE is $\mathcal{O}(|N| \cdot |D|^3)$, where N is the set of nodes of the target program's ICFG and D is the data-flow domain of the analysis.

In addition, IDE allows to annotate the ESG's edges with lambda functions – so-called *edge functions* $f \in J$ – which operate on a separate value domain V and encode an additional value-computation problem. The value-computation problem specified using the ESG edges is solved when performing a reachability check. This way, IDE is able to effectively encode problems with infinite domains such as linear-constant propagation with $D = \mathcal{V}$, where \mathcal{V} is the set of program variables and $V = \mathbb{Z}_\perp^\top$. In this setup, IDE would propagate constant variables through the program and compute their constant values using the edge functions. An exemplary ESG for a linear-constant propagation encoded in the aforementioned manner is shown in Figure 1. The ESG nodes are visualized in a matrix structure where the rows represent the program statements n_1, \dots, n_4 and the columns represent the data-flow facts a, b, p and the special Λ fact. This way, Figure 1 also shows the bipartite nature of the encoded flow functions.

The jump functions constructed by the IDE algorithm describe data flows (and corresponding value computations). They comprise quadruples $\langle d_1, n, d_2, f \rangle$, where $d_1 \in D$ is the data-flow fact that holds at the source instruction (or node in the ICFG) $s_p \in N$, $n \in N$ is the target instruction, $d_2 \in D$ is the data-flow fact at the target instruction, and $f \in J$ is a function that describes the respective value computation. The source instruction s_p is implicit – it is the first instruction of the procedure that is being analyzed. In Figure 1, the jump function that describes that the data-flow fact a holds at ICFG node (n_4) in the program shown thus is: $\langle \Lambda, n_4, a, \lambda \ell. \ell \circ \lambda \ell. \ell + 2 \circ \lambda \ell. 1 \rangle \equiv \langle \Lambda, n_4, a, \lambda \ell. 3 \rangle$. Its evaluation yields that variable a carries the constant value 3 at ICFG node (n_4).

If an ESG node (n, d) is reachable along multiple program paths, the edge functions associated with the respective jump functions are combined using a *join* operation. Similar to flow functions, edge functions must distribute over the *join* operation. Hence, edge functions must be evaluable functions supporting regular function composition as well as the binary *join* operation and an *equality* relation. These operations – and the implementations for the flow and edge functions – need to be specified by analysis writers for the specific data-flow problem at hand.

The number of edges in an ESG is in $\mathcal{O}(|N| \cdot |D|^2)$. Even though D must be finite, D can be very large. Constructing the full ESG can easily lead to a graph containing millions of nodes and edges even for moderately-sized programs. Nearly all open-source state-of-the-art IDE implementations therefore construct only the valid paths reachable from the entry point $(s_{\text{main}}, \Lambda)$ in an on-the-fly manner, as proposed by Naeem et al. [15].

Naeem's on-the-fly algorithm requires the following essential structures to solve an analysis problem:

- *JumpFn* ($D \times N \times D \rightarrow J$): Jump functions $\langle d_1, n, d_2, f \rangle$ tabulated by the IDE algorithm that describe the data-flow facts reachable from $(s_{\text{main}}, \Lambda)$.
- *Incoming* ($N \times D \rightarrow N \times D$): A set that records nodes $\langle s_p, d \rangle$ that the analysis has observed to be reachable and predecessors of $\langle s_p, d \rangle$, where $s_p \in N$ a start point of procedure p . Using this set avoids the need to compute inverse flow functions, which might not be possible for all analysis problems.
- *EndSummary* ($N \times D \rightarrow N \times D \times J$): A table that stores jump functions that summarize the effect of a complete procedure p : $\langle s_p, d_1, e_p, d_2, f \rangle$, where $e_p \in N$ an exit point of p .

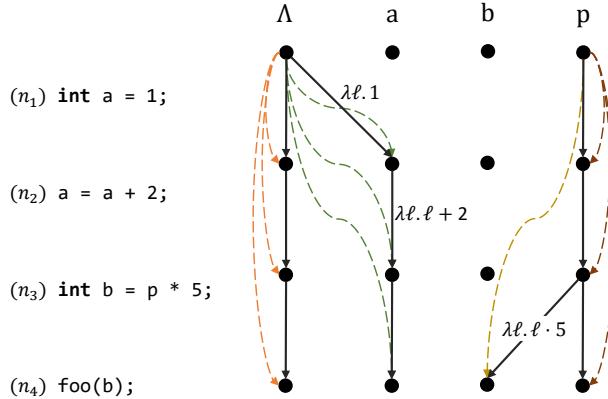


Figure 1 An example exploded supergraph for a linear constant analysis encoded in IDE [17]. The solid edges represent the individual flow functions, whereas the jump functions are denoted by the colored dashed edges. All (solid) flow edges are annotated with their edge functions; identity edge functions have been omitted to avoid cluttering. By following the flow edges in backwards direction, we can see that at (n_4) variable a is reachable from Λ and thus holds as data-flow fact. This information is also encoded as green dashed jump function from (n_1, Λ) to (n_4, a) . Composing the annotated edge functions, we can see that at (n_4) , variable a has the constant value 3.

These per-fact procedure summaries are reapplied in each subsequent context p is called.

2.1 IDE Algorithm Overview

The IDE algorithm works in two phases: (I) Constructing the relevant part of the ESG and (II) computing the values associated to the node-data-flow-fact pairs (n, d) by evaluating all edge functions f annotated to the jump functions in the ESG. We provide a copy of the original IDE algorithm as part of our supplementary website for this paper².

Phase I works as fixed point iteration starting from initial ESG nodes, called *seeds*. Based on the ICFG and the set of flow- and edge functions, the procedure `ForwardComputeJump-FunctionsSLRPs` (see algorithm Phase I) incrementally extends the ESG by adding new edges or updating the annotated edge functions of existing edges. This extending and updating of the ESG is performed by the `Propagate` (see algorithm Propagate) procedure, which gets iteratively called by the solver until a fixed point is reached. The final ESG for the example code snippet in Figure 1 is shown in the same figure (excluding the content of function `foo`).

Phase II (see algorithm Phase II) works in two steps: *value propagation* and *value computation*. First, in the value propagation phase, the initial edge values are propagated iteratively through the ESG from the seeds to the beginning of all analyzed procedures. After that, in the value computation phase, the edge functions of all remaining jump functions are evaluated with the values previously aggregated at the beginning of the respective procedure.

For example, consider the code snippet in Figure 1. Assuming that it is part of a function that gets called with $p = 4$, the value propagation will create the relation $(n_1, p) \mapsto 4$. If the code snippet is called with multiple different values for p , the relation gets updated using the lattice join of the value domain. Further, to aggregate the starting values for all procedures, the value propagation computes the relevant edge values for the call-site, in this case for b at n_4 . It computes $b = (\lambda\ell.\ell \cdot 5)(4) = 20$ and iteratively propagates it into `foo`. After the value-propagation phase has finished, all remaining result relations can be computed, which leads to $(n_2, a) \mapsto 1, (n_2, p) \mapsto 4, (n_3, a) \mapsto 3$, etc.

² Supplementary website: <https://secure-software-engineering.github.io/paper-idesolverxx/>

3 The State of the Art

In many years of developing static data-flow analyses, we have found that state-of-the art analysis implementations, many of them implementing IDE (or a subset of it), do not scale to large programs comprising several hundreds of thousands to millions of lines of code. In the following, we report on the problems with current IDE implementations, with the example of PhASAR, that has lead us to define the optimizations to IDE that we present in Section 4.

To show the performance of a current state-of-the-art IDE implementation, we use the current `IDESolver` from PhASAR³ in version v2403, which is the most recent stable version of the open-source framework at the time. To assess the state-of-the-art, we have applied the `IDESolver` to 31 real-world C and C++ programs⁴ denoted in Table 1 and solved a typestate analysis (TSA), a linear constant analysis (LCA), and an instruction-interaction analysis (IIA) [21]. In Table 1 the columns with the analysis problems are sorted in ascending order by analysis complexity.

Measuring runtime and memory usage of the analysis runs, as Table 1 shows, we observed that, with increased analysis complexity, the number of recorded timeout (t/o) and out-of-memory (OOM) events grows. While the `IDESolver` was able to complete the LCA and TSA on almost all target programs, the solver performed worse on the IIA: In fact, we observed that six out of 31 could not be run on an ordinary developer machine, seven others ran out-of-memory while four others timed out.

The current situation, as illustrated by Table 1, that many interesting data-flow analyses cannot be solved on medium-sized to large target programs is unacceptable. While long runtimes can be tackled by running the analysis less often (e.g., in a CI/CD pipeline) or by increasing the time budget, the high memory requirements are often impossible to solve due to hardware limits; more memory might be integrated which then—depending on the system—would incur high procurement- and operating costs.

As some state-of-the-art IDE implementations, such as PhASAR and Heros, are open-source, we are able to analyze them to gain insights where the performance bottlenecks are and propose optimizations (cf. Section 4) for lowering the time- and memory requirements of IDE.

4 Optimizations

To mitigate the scalability issues of IDE identified in Section 3, we reviewed state-of-the art literature regarding IDE implementations, profiled the IDE solver implementation within the PhASAR framework, and identified two aspects that suggest to offer potential for effective optimizations in terms of both runtime and memory consumption. Although the IDE algorithm works in two phases (see Subsection 2.1), we can tell from our experience that IDE spends the majority of its time during phase I—the part that IFDS and IDE have in common. Thus, we aim to optimize phase I.

First, while computing the target analysis' fixed point, an IDE implementation must efficiently store the set of jump functions. This corresponds to the *JumpFn* map [20] in the original algorithm. The jump-functions table stores all ESG edges that are computed by the IDE solver. That is, it stores quadruples drawn from $(D \times N \times D) \rightarrow J$. The size of the jump-functions table is therefore bound by $\mathcal{O}(|N| \cdot |D|^2)$. As it is unlikely to

³ PhASAR: <https://github.com/secure-software-engineering/phasar/tree/v2403>

⁴ Subsection 6.2 provides details on how the results were obtained and how the analyses were configured.

Table 1 On the left, we see all evaluation targets with additional information, such as the revision we analyzed and the amount of LLVM-IR code. The IR code size is important because PhASAR’s IDE solver works at the IR level. In addition, we report the number of procedures (Proc), the number of globals (Glob), and the number of call-sites (Calls) in the IR, which may influence the performance of the analysis. The three rightmost columns show time [s] and memory consumption [MiB] of the benchmarked analyses utilizing the IDESolver from PhASAR. Orange cells indicate that the memory of a common consumer machine (32 GiB) was exceeded. Dark orange cells indicate that even a compute cluster with 128 GiB would be insufficient. Red cells indicate the analysis ran out-of-memory with a memory limit of 250 GiB, and blue cells represent timeout (t/o) events exceeding four hours of analysis time.

Revision	Domain	LOC	Proc	Global	Calls	Typestate		LCA		IIA	
						Time	Mem	Time	Mem	Time	Mem
FastDownward	641d70b3	Planning	849k	35k	5k	176k	20	1 407	81	7 709	-
asterisk	a0946200	Signal processing	626k	8k	15k	85k	72	4 131	t/o	-	OOM
bison	849ba01b	Parser	123k	1k	1k	13k	38	1 974	82	8 885	-
bitlbee	fb774da0	Chat client	91k	1k	2k	12k	1	203	17	2 126	-
brotli	9801a2c5	Compression	103k	978	173	10k	2	315	9	1 640	505
bzip2	1ea1ac18	Compression	29k	154	182	1k	3	166	20	1 829	842
cat	1913bfcf	UNIX utils	6k	223	139	736	<1	45	1	243	40
cp	1913bfcf	UNIX utils	23k	524	373	3k	<1	86	4	577	288
dd	1913bfcf	UNIX utils	19k	319	287	2k	<1	69	11	1 214	497
file	e94d5264	UNIX utils	1k	66	170	314	<1	39	<1	53	3
fold	1913bfcf	UNIX utils	6k	210	130	715	<1	52	2	245	41
grep	cb15dfa4	UNIX utils	79k	808	424	6k	1	207	25	3 208	545
gzip	23a870d1	Compression	17k	251	351	1k	<1	67	7	1 049	91
htop	bc22bee6	UNIX utils	58k	917	1k	7k	19	290	12	1 647	1 680
hypre	f69f8ef4	Solver	713k	3k	3k	71k	86	6 461	1 259	77 313	t/o
join	1913bfcf	UNIX utils	10k	267	184	1k	<1	66	2	324	55
kill	1913bfcf	UNIX utils	5k	196	135	663	<1	43	1	215	39
lepton	429fe880	Compression	139k	3k	889	24k	3	331	35	4 062	2 902
libjpeg_turbo	2cad2169	File format	142k	582	184	7k	1	242	161	9 172	-
libsigrok	68321f73	Signal processing	148k	1k	4k	16k	2	338	8	1 257	t/o
libzmq	ec6f3b1d	C++ Library	162k	5k	1k	26k	29	2 120	9	901	t/o
ls	1913bfcf	UNIX utils	31k	646	515	3k	<1	111	14	1 642	301
lz4	4a555363	Compression	35k	445	424	4k	12	221	5	847	749
openvpn	cec4353b	Security	187k	3k	4k	24k	10	540	74	8 135	t/o
opus	bce1f392	Codec	131k	851	472	10k	1	233	38	5 160	3 851
poppler	315ab300	Rendering	546k	15k	15k	87k	207	3 573	125	11 788	-
uniq	1913bfcf	UNIX utils	7k	242	181	939	<1	54	2	260	44
wc	1913bfcf	UNIX utils	10k	272	187	1k	<1	61	2	338	52
whoami	1913bfcf	UNIX utils	5k	180	113	539	<1	42	1	209	36
x264	e067ab0b	Codec	500k	2k	2k	33k	48	3 151	203	19 605	-
xz	e7da44d5	Compression	10k	252	455	1k	<1	57	2	327	31

reduce this worst case bound, we propose in Subsection 4.1 to lower the constant factors of these bounds by optimizing the memory layout of the jump-functions table, which enables practical performance gains. Second, most jump functions computed by IDE are just needed temporarily to craft the procedure summaries ψ . Once a summary has been created, the corresponding intermediate jump functions are no longer needed. Hence, to reduce IDE’s memory footprint, we propose in Subsection 4.2 to remove such intermediate entries from the jump-functions table. In fact, we extend the work from Arzt [1] by designing a garbage collector for jump functions that—in contrast to the one proposed by Arzt—is applicable to arbitrary IDE problems.

It is important to note that our optimizations do not target just one particular implementation; our optimizations are generally applicable.

4.1 Data Structures for the Exploded Supergraph

While solving an IDE data-flow analysis problem, the solver incrementally creates jump functions (see Section 2) that need to be stored in memory. To solve the analysis problem efficiently, the jump functions need to be stored efficiently, allowing for short lookup and insertion times as well as for a small memory footprint.

$(d_1, n, d_2) \mapsto f$
$(\Lambda, n_1, \Lambda) \mapsto \lambda \ell. \ell$
$(p, n_1, p) \mapsto \lambda \ell. \ell$
$(\Lambda, n_2, \Lambda) \mapsto \lambda \ell. \ell$
$(p, n_2, p) \mapsto \lambda \ell. \ell$
$(\Lambda, n_2, a) \mapsto \lambda \ell. 1$
$(\Lambda, n_3, \Lambda) \mapsto \lambda \ell. \ell$
$(p, n_3, p) \mapsto \lambda \ell. \ell$
$(\Lambda, n_3, a) \mapsto \lambda \ell. 3$
$(\Lambda, n_4, \Lambda) \mapsto \lambda \ell. \ell$
$(p, n_4, p) \mapsto \lambda \ell. \ell$
$(\Lambda, n_4, a) \mapsto \lambda \ell. 3$
$(p, n_4, b) \mapsto \lambda \ell. \ell \cdot 5$

Figure 2a. The jump-functions table similar to the *FastSolver* of FLOWDROID. Without nesting, the whole jump functions $\langle d_1, n, d_2, f \rangle$ of the ESG for Figure 1 are stored in one level which may lead some of d_1 , d_2 , and n being stored redundantly.

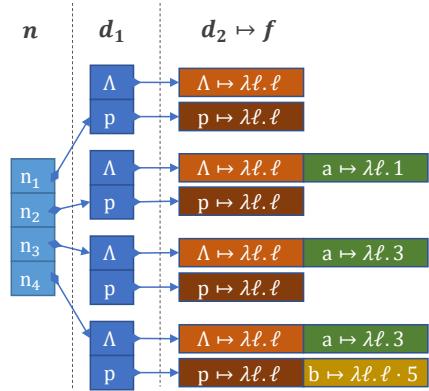


Figure 2b. The main jump-functions table from PhASAR and Heros. For each jump function $\langle d_1, n, d_2, f \rangle$, it maps the nodes n to inner maps, which map the source data-flow facts d_1 to the respective target facts d_2 and edge functions f . This avoids some nodes n and source facts d_1 to be stored multiple times, as they would be in Figure 2a, but adds extra cost for the inner mappings.

▀ **Figure 2** Different jump-functions table layouts currently used by open-source IDE implementations.

4.1.1 Jump Functions Table Analysis

Existing IDE solver implementations such as Heros [5], PhASAR [22] and FLOWDROID [4] use different representations to store jump functions, each of which comes with different performance properties. PhASAR and Heros use nested mappings $N \rightarrow (D \rightarrow (D \rightarrow J))$ that map a target node $n \in N$ to a map of source data-flow fact $d_1 \in D$ to a map of target fact $d_2 \in D$ to the associated edge function $f \in J$. Yet, to speed up algorithm-specific lookup and insert tasks, Heros and PhASAR store each jump function redundantly in two additional maps, effectively modeling a multi-index table. In what follows, when referring to the jump-functions table structure used by PhASAR and Heros, we focus on the nested mapping described above, but keep in mind that the multi-index may have a drastic impact on the overall memory consumption of the solving process.

FLOWDROID uses a flat $(N \times D \times D) \rightarrow D$ representation to map a full jump function $(n, d_1, d_2) \in N \times D \times D$ to the same target fact d_2 . As FLOWDROID only implements IFDS, which is a subset of IDE where all edge functions are implicitly the identity function $\lambda x.x$, it does not need to store edge functions $f \in J$. It stores the target fact twice for implementation-specific support for path-tracking. As path tracking is out of scope for this work, we concentrate on the $(N \times D \times D)$ part of the data structure.

Both data structures (nested and flat) have their advantages and drawbacks. Consider the example in Figure 1. Having no nested mappings, as shown in Figure 2a, makes lookup and insertion fast, since they only consist of a single hash-map operation. In contrast, the nested approach, as shown in Figure 2b, requires three hash-map operations for each lookup or insert as for each of n , d_1 and d_2 in a jump-function entry a separate hash-map lookup or insertion is required.

In both designs, the noticeable duplication of the edge functions f could be solved by storing them in a separate cache. PhASAR, in fact, supports such a cache already. However, even with caching edge functions, nodes n and source facts d_1 may be stored redundantly in memory. This is, because it is likely that there are multiple jump functions

that lead to the same target node, which corresponds to the existence of the jump functions $(d_{1,1}, n, d_{2,1}), \dots, (d_{1,k}, n, d_{2,m})$ for $n \in N$, $\{d_{1,1}, \dots, d_{1,k}, d_{2,1}, \dots, d_{2,m}\} \subseteq D$ and $k, m \in \mathbb{N}$. Such jump functions may store the target node n multiple times in a flat structure, such as Figure 2a, but store n only once in a nested representation such as Figure 2b.

In the same vein, when generating data-flow facts, it is also likely that there are multiple target facts for the same source-fact and target node, for example, jump functions of the form $(d_1, n, d_{2,1}), \dots, (d_1, n, d_{2,m})$ for $n \in N$, $\{d_1, d_{2,1}, \dots, d_{2,m}\} \subseteq D$ and $m \in \mathbb{N}$. For instance, the jump functions $(\Lambda, n_2, \Lambda, \lambda \ell. \ell)$ and $(\Lambda, n_2, a, \lambda \ell. 1)$ from Figure 1 fall in that category. In a flat representation such as of Figure 2a, jump functions store both source fact d_1 and target node n redundantly, but avoid the redundant storage in a nested representation as shown in Figure 2b.

In summary, nested mappings store less data from the jump functions redundantly and therefore are likely to expose a lower memory usage than a shallow representation. Conversely, common operations such as lookup and insertion of jump functions in the table are likely to be faster in the flat representation as there are fewer indirections and fewer hashing operations. Furthermore, map data structures themselves have implementation-specific memory overhead. Therefore, a nested representation is more memory efficient than a flat one only if the additionally introduced maps grow beyond an implementation-specific threshold to compensate the overhead of these maps.

4.1.2 Optimized Jump Functions Table

Given the analysis in Subsubsection 4.1.1, we propose a compromise between nested and flat data structure representations that harnesses the advantages of both to drastically improve both the memory usage as well as the runtime of the IDE algorithm. We acknowledge that a nested mapping is necessary to avoid duplicate storage of nodes and data-flow facts. However, to keep lookup times low and to keep the individual maps sufficiently large, we aim at reducing the nesting depth as well. Specifically, we propose a two-level nested map as a compromise between fast lookup times and low memory usage. For a design with two levels of nesting, there are six possible mappings to store jump functions:

- | | |
|---------------------------------------|-------------------------------------|
| 1. $(n, d_1) \mapsto (d_2 \mapsto f)$ | 4. $n \mapsto (d_1, d_2) \mapsto f$ |
| 2. $(n, d_2) \mapsto (d_1 \mapsto f)$ | 5. $d_1 \mapsto (n, d_2) \mapsto f$ |
| 3. $(d_1, d_2) \mapsto (n \mapsto f)$ | 6. $d_2 \mapsto (n, d_1) \mapsto f$ |

To reduce the number of candidate representations, we consider one more optimization: As we limit ourselves to two-level nested maps, each jump functions access requires at least two indirections. However, with intelligent batch-processing, the effective number of indirections can be reduced. We observe that during ESG construction in the IDE algorithm(cf. Subsection 2.1), the only direct access to the jump-functions table is inside the `Propagate` function depicted on the left side of Algorithm 1. Here, the expression $JumpFn(e)$ performs the jump-functions table access where e represents a complete jump edge consisting of the target node n and the source- and target data-flow facts d_1 and d_2 . We further observe that in the original algorithm `Propagate` is always called from within a loop where parts of n , d_1 , or d_2 are loop-invariant.

So, if we design the jump-functions table accordingly, we can optimize the `Propagate` procedure (shown on the right side of Algorithm 1), by batching the access to the outer map for multiple jump functions accesses together. Here, `Propagate` receives an additional parameter j that denotes a view into the jump-functions table where the loop-invariant parts

Algorithm 1 The modifications in the `Propagate` procedure that support batch processing. An exemplary use of `Propagate` for the case in which the target node n is loop-invariant is shown in Lines 8-11. To highlight changes compared to the original algorithm from Sagiv et al. [20], additions are shown in **green** and removals are shown in **red**.

<pre> 1 Procedure Propagate(e, f) 2 let $f' = f \sqcap \text{JumpFn}(e)$; 3 if $f' \neq \text{JumpFn}(e)$ then 4 $\text{JumpFn}(e) = f'$; 5 Insert e into <i>PathWorkList</i>; 6 end 7 end // Example use: 8 9 for ... do 10 Propagate($\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle, f$); 11 end </pre>	<pre> Procedure Propagate(j, e, f) let $f' = f \sqcap j(e)$; if $f' \neq j(e)$ then $j(e) = f'$; Insert e into <i>PathWorkList</i>; end // Example use: j = JumpFn($\langle *, * \rangle \rightarrow \langle n, * \rangle$); for ... do Propagate($j, \langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle, f$); end </pre>
--	---

Table 2 Access patterns of the jump-functions table with their number of occurrences within the original IDE algorithm [20] (cf. Subsection 2.1).

Invariant parts	# Occurrences
n	1 (call-flow)
n, d_1	2 (call-to-return-flow, summary-flow)
n, d_2	1 (return-flow)
d_1	1 (normal-flow)

are already fixed. In the example, j only contains jump functions where the target node is a previously fixed n . It is important that the extraction of j happens outside of the loop that calls `Propagate`. Using the smaller map j for accessing the jump functions instead of the complete table JumpFn may improve the performance of `Propagate`. In fact, if j is one of the inner maps of our two-level nested jump-functions representation, using j effectively reduces the nesting depth of the table within `Propagate`, which in turn reduces the runtime cost of accessing individual jump functions.

Efficiently extracting the view j from the jump-functions table requires that the jump-functions table is laid out in a way that supports this operation. This can be achieved by placing the loop-invariant parts as keys into the outer map and the loop-variant parts into the inner maps. To decide which view j is best suited to achieve maximum performance improvement, we have to analyze which parts, n , d_1 , or d_2 , of a jump function are most frequently loop-invariant.

Based on careful analysis of the original algorithm [20], we identify four different access patterns, as depicted in Table 2. Although n is not strictly invariant in the normal-flow case, it may still be beneficial to consider n as invariant for the purpose of selecting a jump-functions representation, as most intraprocedural control-flow nodes mostly have only one (statement-sequence) or two (conditional branch) successors. Furthermore, to propagate all normal flows, the algorithm needs to iterate over all relevant n, d_2 pairs which is usually implemented as nested loop, effectively making n or d_2 temporarily loop-invariant. This consideration has no influence on the algorithmic correctness, but on the effectiveness of batch-processing jump functions accesses in the table.

Based on these observations, we conclude that it is beneficial to store the target fact d_2 in the inner map and n in the outer map. This enables us to filter out most of the six possible mappings presented above, leaving only

$$\begin{array}{ll} \text{1. } (n, d_1) \mapsto (d_2 \mapsto f) & \text{4. } n \mapsto (d_1, d_2) \mapsto f \end{array}$$

as possible candidates, which we call JF_{ND} and JF_N , respectively, denoting the domain used in the outer map.

Furthermore, we also conjecture that a multi-index representation of the jump-functions table is not necessary. With any of JF_{ND} or JF_N we can efficiently model all access patterns that occur in the IDE algorithm. Hence, we introduce a third jump-functions representation, JF_{old} , that uses the deep nesting from PhASAR and Heros ($n \mapsto d_1 \mapsto d_2 \mapsto f$), but avoids the multi-index.

Our theoretical analysis also yields that, with JF_{ND} , we already have efficient access to the procedure summaries, eliminating the need for an extra *EndSummary* table that was proposed by Naeem et al. [15]. To access a summary⁵ of procedure p , we can directly lookup the necessary jump functions at p 's exit statements. With JF_N , to find matching summaries without the *EndSummary* table, one requires a linear search over the inner maps at p 's exit statements. Depending on the size of these inner maps, this linear search may still be fast, so we split JF_N into two candidates: JF_N and JF_{NE} where JF_{NE} uses the explicit *EndSummary* table while JF_N omits it.

4.1.3 Discussion

From the observations in Subsubsection 4.1.2, one could conclude that JF_{ND} is superior to JF_N because, in three out of the five `Propagate` calls, d_1 is loop-invariant. However, in JF_{ND} (depicted in Figure 3a) the outer map is larger than in JF_N (depicted in Figure 3b) as its key space is larger: $|N| \leq |N \times D|$. Therefore, JF_{ND} needs to store more inner maps than JF_N although, in the end, both store the exact same number of jump functions. Furthermore, the inner maps in JF_{ND} are smaller than the inner maps in JF_N , as there are more of them and depending on the concrete implementation-specific overhead of a single inner map, the memory cost of the inner maps might outweigh their potential benefit. Hence, from a sole theoretical analysis, we cannot conclude which jump-functions representation performs better in practice; we need to perform an empirical evaluation to draw a final conclusion (Section 6).

4.2 Garbage Collection of Jump Functions

As discussed in Subsection 4.1, the jump-functions table has a great influence on the overall memory consumption of the IDE algorithm. Arzt [1] has shown that it is possible to remove entries in the jump-functions table without preventing the algorithm from reaching a fixed point. They present a garbage collector (GC) that runs concurrently to the actual IDE implementation, improving both memory usage and runtime of the underlying analysis. The GC removes jump functions when they are no longer needed. This applies when the complete data flow represented by a jump function has already been composed to a summary.

One limitation of the approach of Arzt [1] is that it only applies to an IFDS analysis and therefore does not need to deal with edge functions. In IDE, the value computation problem on data-flow edges can only be performed if the corresponding jump functions are

⁵ Processing summaries as described in line 15.2 by Naeem et al. [15].

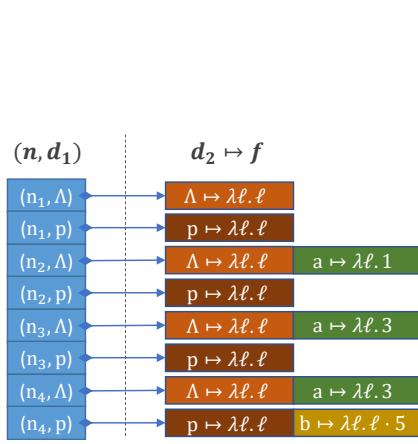


Figure 3a. jump-functions representation JF_{ND} for the example shown in Figure 1. The outer map has a two-dimensional key space consisting of the target node n and the source fact d_1 , which reduces the size of the inner maps, containing only the target fact d_2 and the edge function f .

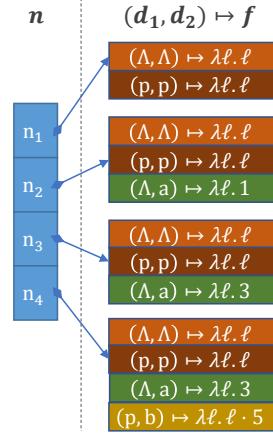


Figure 3b. jump-functions representation JF_N for the example shown in Figure 1. The outer map has a one-dimensional key space only consisting of the target node n , whereas the inner maps have a two dimensional key space containing the source- and target facts d_1 and d_2 as well as the associated edge functions f . Compared to JF_{ND} , JF_N contains fewer inner maps which in turn grow larger.

■ **Figure 3** Exemplary jump-functions tables using the proposed representations JF_{ND} and JF_N .

present. This makes garbage collecting jump functions more complicated in a general IDE setting with associated edge functions. Although Arzt describes a possible extension of the GC to IDE as trivial, we recognize that the correct handling of corner cases makes it less obvious than it seems on the first glance. Especially, we need to ensure that subsequent result queries can still evaluate the edge-functions correctly that are annotated to the jump functions. Secondly, the garbage collection by Arzt [1] exploits multithreading at the level of the data-flow analysis solver. This requires the complete analysis toolchain to be thread safe. While some IDE implementations do satisfy this requirement and make use of multiple cores to speedup the solving process, other implementations are only single-threaded and do not provide thread-safe data structures. Specifically, PhASAR’s analyses are not thread-safe and even LLVM—which PhASAR builds upon—is not generally thread-safe. Additionally, since we conduct a comprehensive study evaluating the runtime and memory consumption of IDE, we need to ensure that external factors, such as OS scheduling do not influence our evaluation results. Hence, we prefer using only a single thread, which eliminates many of these issues by removing non-determinism from the implementation.

In the following, we describe how we mitigate both limitations, the restriction to a subset of IDE and the enforced multi-threading.

4.2.1 Single-Threaded Garbage Collection

To keep the GC scalable, Arzt designed it to work on a procedure-level. That is, all jump functions corresponding to procedure p can be erased once there is no longer any worklist item that contains a node from inside p or from any procedure that can be transitively called by p [1]. We call this the *GC Condition*. Unfortunately, the order in which the ESG is constructed is not specified by the underlying algorithm [20], which is why one cannot precisely predict these points. If the garbage collector runs concurrently to the

actual analysis-solving thread, it can be invoked periodically based on a timer. Additional computations that the GC needs to perform to determine for which procedures the jump functions can be erased do not necessarily pause the analysis. However, as explained above, we decided to aim for a single-threaded solution here. The GC thus needs to be called explicitly at suitable points within the IDE algorithm and will pause the data-flow analysis for the garbage collection.

We observe that a procedure p can only become a candidate for garbage collection once the analysis within p has reached an exit statement. In theory, it is possible to invoke the GC after exiting any procedure, yet this has a non-negligible overhead that would render the analysis unscalable. Hence, we aim for finding a point in the IDE algorithm to place the GC, such that it gets called frequently enough to keep it effective, but not too frequent to keep it scalable. This means, that the GC should be invoked, once a sufficient amount of procedures have computed their summary.

There are several ways of deciding when the GC should be invoked, each with different characteristics and implications. One approach is to increment a counter, whenever a procedure has computed a new summary, and invoke the GC when the counter reaches a certain threshold. This approach has the advantage that it is easy to implement. On the downside, it does not decide to invoke the GC based on concrete information on the internal solver state, such as the content of the worklist or the jump-functions table. Therefore, many candidate procedures may actually fail the *GC Condition* and are not eligible for garbage collection yet. Hence, its performance may not be predictable and requires a decent amount of tuning. An alternative is to take the contents of the solver's worklist into account when deciding on when to invoke the GC. Since the *GC Condition* is based on the content of the worklist, we can invoke the GC when it is guaranteed that the candidate procedures will pass the *GC Condition*. In our implementation, we opted for this more informed procedure.

For deciding, when to invoke the GC, we split IDE's worklist into two separate worklists: One *Path WorkList* for top-down propagations, which stores jump functions in $D \times N \times D \times J$ to be processed, and another worklist, *Ret Worklist*, for bottom-up summary applications that stores entries of the form $(d_1, p) \in D \times P$, where P is the domain of callable procedures in the target program. On a high level, the fixed-point iteration uses the *Path WorkList*, but also fills the *Ret Worklist* on-the-fly when a procedure has reached its exit point. Once the *Path WorkList* becomes empty, the algorithm handles the work-items from the *Ret Worklist*, which may fill the *Path WorkList* again. Although the data-flow propagations have stayed the same, using two worklists we now have structured the fixed-point iteration into stages (a stage ends, whenever the *Path WorkList* becomes empty) that allow placing a call to the GC.

For the two worklists to function properly, we modify the IDE algorithm as sketched in Algorithm 2. The pseudo code for handling procedure exit points that we removed in Line 9 of Algorithm 2 has moved to a new outer loop depicted in Algorithm 3. As applying procedure summaries may lead to new intra-procedural propagations at their return sites, the whole process runs in a loop until *both* worklists are empty, as shown in Algorithm 3.

Note that in subsequent iterations, the `ForwardComputeJumpFunctionsSLRPs` procedure must skip its initialization phase to not over-write the already computed results. Apart from that, we did not change the original IDE algorithm, as we describe in Paragraph 4.2.1.1.

Using two worklists, the garbage collection condition now slightly changes. The jump functions of a procedure p can only be collected if none of the *Path WorkList* and the *Ret Worklist* contain a node from inside p or its transitive callees. This is, because when processing the worklist items (d_1, p) from the *Ret Worklist*, the callers of p may be added to the *Path WorkList* again preventing garbage collection for p . Whenever the *Path WorkList* is

■ **Algorithm 2** Modification in the `ForwardComputeJumpFunctionsSLRPs` procedure from the original IDE algorithm [20].

```

1 Procedure ForwardComputeJumpFunctionsSLRPs(... )
2   ...
3   while PathWorkList ≠ ∅ do
4     Select and remove an item  $\langle s_p, d_1 \rangle \rightarrow \langle n, d_2 \rangle$  from PathWorkList;
5     ...
6     switch n do
7       ...
8       case n is the exit node of p do
9         | Insert  $(d_1, p)$  into RetWorklist;
10      end
11      ...
12    end
13  end
14 end

```

■ **Algorithm 3** High-level overview of the two-step fixed point computation with garbage collection. The `foreach` loop in Line 5 denotes the content from `ForwardComputeJumpFunctionsSLRPs` [20] that we have removed from Algorithm 2. The function `RunGarbageCollector` behaves exactly as described by Arzt [1].

```

1 while PathWorkList ≠ ∅ do
2   ForwardComputeJumpFunctionsSLRPs(... );
3   while RetWorklist ≠ ∅ do
4     Remove  $(d_1, p)$  from RetWorklist;
5     foreach call node c that calls p with corresponding return-site r do
6       ...
7     end
8   end
9   RunGarbageCollector();
10 end

```

empty, we have the guarantee that for all currently analyzed procedures (and their transitive callees), the analysis has reached their exit points, making them candidates for garbage collection. Hence, we now have a structure that precisely defines points for placing the GC.

In particular, we now have two candidate locations to place the garbage collection in Algorithm 3: Line 3: Right after the returning from `ForwardComputeJumpFunctionsSLRPs` (i.e., when the `PathWorkList` becomes empty) or Line 9: After the `RetWorklist` becomes empty. In Line 3, the `RetWorklist` is potentially non-empty as it may contain procedures p that have computed a new summary for the propagation of a source data-flow fact d_1 that needs to be propagated back to all callers of p . In Line 9, though, the `RetWorklist` is empty, whereas the `PathWorkList` may be filled with return flows again.

Both insertion points at Line 3 and Line 9 are very similar, however, Line 9 has one small benefit: Having a jump function from a procedure p in the `RetWorklist` prevents all caller procedures of p from being garbage collected. After processing the `RetWorklist` items, only those callers of p have jump functions in the `PathWorkList` for which the new information

from p requires further propagation. All other caller procedures can still be garbage collected (unless there are other callees that prevent the collection). This leads to our preference to place the garbage collection at Line 9. Note, although the worklists are processed until completion in one iteration of the outer loop, there are still potentially many iterations such that the garbage collector is run many times as well.

4.2.1.1 Correctness

Our modifications to the IDE algorithm and the integration of the garbage collection do not violate the correctness and complexity of the IDE algorithm. Splitting the worklist into two smaller worklists, as we have done in Algorithm 2 and Algorithm 3, does not create new worklist items that would not be created in the original, and also does not drop worklist items that would be processed in the original. Only the order, in which the worklist items are processed, may change. This is, because (1) the processing of exit nodes (cf. Line 9) gets delayed through the *Ret Worklist* to Algorithm 3 without modifying the corresponding worklist items, and (2) since the processing order of the worklist items is not defined in the algorithm [20], any modification on the processing order has no influence on the correctness or complexity of the algorithm.

In addition, we use the same *RunGarbageCollector* function from Arzt without modification. Only the garbage collection condition, has slightly changed: Whereas in the original GC, a procedure p 's jump functions can be erased, if the worklist does not contain a node from inside p or its transitive callees, in our extension, this requirement holds for both the *PathWorkList* and the *Ret Worklist*. Since we argue above that both *PathWorkList* and *Ret Worklist* in combination express the same worklist items as the original worklist, the correctness argumentation from Arzt still holds.

4.2.2 Generalizing Garbage Collection for IDE

When a procedure p gets evicted by the original GC from Arzt, all jump functions corresponding to that procedure are removed. However, when performing an analysis that uses IDE's edge functions, one needs to ensure that the value computation (cf. Subsection 2.1) can still be performed correctly. To solve the value computation problem for an ESG node $(n, d) \in N \times D$, the edge functions annotated to all jump functions that lead to node (n, d) have to be evaluated and thus need to be present. For example, removing the intermediate jump function $\langle \Lambda, n_3, a, \lambda\ell.3 \rangle$ in Figure 1 would prevent that the analysis computes the result relation $(n_3, a) \mapsto 3$. This makes garbage collection for IDE's jump functions impossible when the values for all ESG nodes must be computed. Fortunately, many analyses can define for which ICFG nodes $n_i \in N$ analysis-result queries may be raised before starting the solving process. For example, in a typestate analysis, only the API call nodes that are relevant for the analyzed usage pattern may be queried. We call those nodes n_i *interesting*. At *interesting* nodes, we erase no jump functions in the GC to ensure that at those nodes the complete analysis results including edge values will be present.

However, we have to retain additional jump functions: The value-propagation phase (cf. Subsection 2.1) first propagates initial edge values from the entry points to the starting nodes of all reachable procedures. This is done by iteratively querying and evaluating the jump functions at all call sites to map the initial values to the start of all reachable procedures. This initial value-propagation is necessary for the other jump functions to be evaluated, as it determines the input values for these jump functions. Therefore, for the value propagation to work properly, one must also retain the jump functions at all call sites, even if they are

not considered *interesting*, such that the value propagation to the starting points of all procedures can succeed. Hence, when using IDE’s edge functions, the garbage collection must retain more jump functions than just the ones corresponding to *interesting* nodes, making it potentially less effective.

In the evaluation, we demonstrate that the garbage collection is still effective in a real world setting, even in a single-threaded environment and when using IDE without restrictions.

5 Implementation

We implemented the IDE algorithm including the optimizations proposed in Section 4 on top of the PhASAR framework [22]. PhASAR is able to analyze LLVM IR [13] in a fully automated manner and already provides an implementation of IDE, called **IDESolver** [22, 23]. The **IDESolver** is parametrizable with an user-defined description of an IDE analysis problem that shall be solved. After solving the analysis problem, the **IDESolver** can answer queries about which data-flow facts hold at a given ICFG node and which edge value has been computed for a given node–data-flow fact pair $(n, d) \in N \times D$. We chose to provide the same interface in the new solver such that it can be used as a drop-in replacement. Note that the determination of *interesting* nodes for the garbage collector is completely opt-in, so only IDE analyses that use both the garbage collector and edge functions may need to implement it. We call our new solver **IDESolver++**.

The existing solver provides several configuration options that influence how the analysis problem should be solved (e.g., whether the value computation in IDE Phase II should be performed). Our new implementation is configurable as well, but we chose to lift the configuration from runtime to compile-time. This allows to specialize the solver for the selected configuration such that the algorithms and data structures can be selected precisely for the requested needs. For example, if the implementation detects at compile-time that the to-be-solved analysis problem does not need edge functions, the jump functions table will replace its inner map by a set, eliding the storage for associated edge functions that would otherwise all default to the identity function $\lambda x.x$.

In Section 4, we have shown different representations of the table storing the jump functions, and we concluded that this representation is critical for optimal performance of the overall solving process. Therefore, we chose to use open-addressing⁶ hash maps to store the concrete mappings of the structures JF_{ND} and JF_N , as well as JF_{old} . Open-addressing hash maps are particularly performant because of their cache efficiency and small number of dynamic memory allocations. However, their performance degrades with increasing size of the entries to store. The domains N and D are user defined for both solvers (the current **IDESolver** and our **IDESolver++**) making them generic over the program representation to analyze and the type of data-flow facts. Therefore, we do not use these types directly as keys and values in the hash maps to guarantee predictable performance. Instead, we chose to introduce an intermediate layer that maps each used node and data-flow fact to 32-bit integers in the contiguous ranges $[0, \dots, |N| - 1]$ and $[0, \dots, |D| - 1]$. These integers are then used as keys/values in the actual jump-functions table. The sizes of the intermediate maps are negligible compared to the size of the jump-functions table. We reasonably assume that both N and D do not grow larger than $2^{32} - 1$, since these domains are bound by the size of the input program. For the JF_N (and JF_{NE}) approach, the intermediate layer enables one more optimization: The outer map can be replaced by a plain array to further reduce the memory footprint and to improve lookup performance.

⁶ Open-addressing hash tables store all buckets in a contiguous block of memory, using probing for collision resolution.

Since the inner maps are very small in many cases, we chose to use `llvm::SmallDenseMap<K,V,4>` for the inner maps to optimize for the case in which these maps do not exceed a capacity of 4. This optimization is critical, especially for JF_{ND} and JF_{old} , because they store a large number of small inner maps, where their sizes mostly do not exceed the initial capacity (48 entries) of a regular `llvm::DenseMap`. Independent from the selected jump-functions representation, the corresponding outer hash map is pre-allocated with a reasonable size that scales linearly with the size of the input program. Together with the small-size optimization, this pre-allocation reduces the total number of potentially expensive (re-)allocations.

Our implementation is openly available in the supplementary material of this paper and we are already in contact with the maintainers of PhASAR for rapid integration into the open source framework.

6 Empirical Study

To empirically evaluate the optimizations proposed in Section 4, we use our IDE implementation (see Section 5) to analyze 31 real-world C/C++ programs. We start with defining our research questions.

6.1 Research Questions

Jump-Functions Table Structure

In Subsection 4.1, we have argued that the structure of the jump-functions table directly influences the performance of the analysis, especially regarding memory consumption. Hence, we ask:

RQ₁ | *What is the influence of choosing one of the proposed data structures, JF_{ND} , JF_N , and JF_{NE} , in terms of runtime and memory consumption when analyzing real-world C/C++ programs?*

Jump-Functions Garbage Collection

Arzt [1] has shown that a garbage collector for jump functions not only significantly reduces memory usage of the underlying analysis, but reduces runtime as well. As we have applied significant changes (cf. Subsection 4.2) to the garbage collection by extending it to general IDE problems and mitigating its restriction to multi-threaded analyses, we ask:

RQ₂ | *How effective is the jump functions garbage collector in reducing memory usage and running time when analyzing real-world C/C++ applications without the restrictions to a subset of IDE and a multi-threaded implementation?*

6.2 Experiment Setup

To ensure that our experiments are easily reproducible and comprehensible, we detail on our setup in the following. In Subsubsection 6.2.1, we define what kind of analyses we consider during the evaluation, and in Subsubsection 6.2.3 we present how we perform our measurements as well as the required actions to answer the research questions.

6.2.1 Analysis Problems

To test our solver implementation, we choose to evaluate it using three commonly used analysis problems that put a different amount of load to the solver:

- **TSA:** *Typestate analysis*, configured to find invalid usage patterns of `libc`'s file-IO API
- **LCA:** *Linear constant analysis*
- **IIA:** *Instruction-interaction analysis*, to generate git-blame reports [21].

These analysis problems are available within PhASAR, and we use them unchanged. The *typestate analysis* is expected to put low load on the solver as many programs use `libc`'s file-IO only in few small regions of their code. The *linear constant analysis* should put medium load on the solver, as it needs to propagate all potentially constant integer values; however, the implementation in PhASAR currently is not alias aware, so the load on the solver is still less than for the instruction-interaction analysis, which propagates all potential aliases of the generated data-flow facts. Finally, the *instruction-interaction analysis* puts an extreme load on the solver as it needs to exhaustively track *all* of the target program's variables and capture their interactions with the program's instructions [21]. This way, the size of the data-flow domain D approaches $|N|$ allowing us to approximate the worst-case scenario for field-insensitive analyses.

6.2.2 Target Programs

To ensure that our evaluation results reflect real-world analysis usage as closely as possible, we carefully select the set of 31 target programs shown in Table 1. We select the target programs out of 12 different domains to achieve broad coverage. Further, we choose the target programs in various sizes in the range from 1 676 to 849 623 lines of code in LLVM IR to test the IDE solver with different loads. The target programs have varying properties, such as the number of procedures (66 to 35 134), the number of address-taken functions (0 to 2 696), the number of globals (113 to 15 108), the number of call-sites (314 to 176 350), the number of indirect call-sites (0 to 2 155), and the number of basic-blocks (266 to 111 521).

We include the benchmarked programs from the initial PhASAR paper [22] excluding PhASAR itself, because it has grown significantly since 2019, such that expensive analyses, e.g., the IIA, do not work on that large programs anymore. Still, our evaluation results cannot be compared to the results from Schubert et al. [22], since we use different client analysis problems; the taint analysis used by Schubert et al. is of less interest for our work, since it does not require IDE to be solved efficiently. We also include programs from the evaluation of Sattler et al. [21] as they explicitly report performance problems of PhASAR's IDE solver on their benchmark. In contrast to the PhASAR benchmark, the time and memory results for the programs analyzed by Sattler et al. can be compared to our evaluation results, because the implementation and configuration of the IIA has not changed.

6.2.3 Measurement Setup

Each individual experiment is performed separately for each analysis problem. As analysis targets we use 31 real-world C/C++ programs, which we compile to LLVM 14 IR using WLLVM⁷, so that PhASAR's analyses can consume them. To reduce measurement bias, we run each experiment (solver configuration \times analysis problem \times analysis target) three times and report average values. To validate that our experiments indeed show low variance, we compute the standard deviation of the runtime measurements of the three repetitions. We observe an average standard deviation of 2.2s to 8.3s depending on the jump-functions representation. Normalizing that by the total runtime, the average standard deviation lays

⁷ WLLVM: <https://github.com/travitch/whole-program-llvm>

between 0.99% and 1.5% of the measured runtime. As we expect running times in the area of hours instead of seconds, the impact of measurement bias, as well as the variance between repetition is expected to be negligible and therefore, we consider the relative small number of repetitions $k = 3$ as sufficient to achieve reliable results.

We use the UNIX `time` utility to measure the total runtime and peak memory usage for all experiments. We compute speedups for runtime and memory consumption (maximum resident set size) by comparing the statistics of the to-be-evaluated configuration of the **IDESolver++** to the statistics of the respective baseline. Given runtime measurement samples $M_N = \{m_{n_1}, \dots, m_{n_k}\}$ and baseline-measurements $M_B = \{m_{b_1}, \dots, m_{b_k}\}$ with the number of samples $k = 3$, the speedup is defined as

$$S = \frac{1}{k^2} \sum_{(m_n, m_b) \in (M_N \times M_B)} \frac{m_b}{m_n}$$

For memory measurements, we use the inverse $\frac{1}{S}$ of the above formula to compute the relative memory usage in percent. We compare each combination of m_n and m_b , as these samples are unordered. This prevents potential biases due to sample ordering. Note that in contrast to Arzt [1] we can make use of the external tool `time` for measuring memory consumption, because our experiments do not run in the JVM that makes external memory measurements unreliable.

We conducted our evaluation on a compute cluster in an isolated and controlled environment to ensure that our measurements are not influenced by external factors. Each compute node is equipped with an AMD EPYC 72F3 8-Core processor and 250GiB of RAM, running a minimal Debian 10.

In addition, to increase the reproducibility of our results, we automate the evaluation process with the VaRA Tool-Suite⁸.

Baseline. We also evaluate the existing state-of-the-art **IDESolver** that is openly available in PhASAR as shown in Section 3. As a baseline for our further experiments, we use the **IDESolver++** with the deeply nested jump-functions representation JF_{old} , which the **IDESolver** uses as well. In addition, we compare the both solvers in terms of runtime and memory consumption to assess the influence of our implementation in comparison with the current state-of-the art, when *not* applying the optimizations proposed in Section 4. Note that we do not implement the multi-index table for storing jump functions since the **IDESolver++** does not need it, as discussed in Subsubsection 4.1.2. To achieve a fair comparison, we need to configure the **IDESolver**. We set the configuration option `recordEdges` to `false` to avoid storing the ESG edges in a path sensitive way. We record runtime and memory usage, as well as out-of-memory (OOM) and timeout events of both solvers, providing a baseline to compare against in the evaluations of our research questions.

RQ₁. We evaluate four configurations of our **IDESolver++**, one using JF_{ND} , JF_N , JF_{NE} , and JF_{old} as jump-functions table respectively. JF_{old} serves as a baseline for the others. To judge which jump-functions table structure performs best on our target programs, we compute the speedups compared to the baseline and consider the configuration with the highest speedup as best. To verify whether the best configuration is *significantly* best, we perform a *t*-test with significance level $\alpha = 0.05$. The garbage collector is turned off.

⁸ VaRA Tool-Suite: <https://vara.readthedocs.io/en/vara-dev/>

RQ₂. We configure the **IDESolver++** as follows: turning the GC on or off and using JF_{ND} or JF_N . The **IDESolver++** with GC turned off is used as baseline. We exclude JF_{NE} here, because it stores the jump functions in exactly the same way as JF_N , just with one additional table that only contains jump functions which cannot be evicted by the GC at all. So, in total, we have four configurations for this experiment. For the typestate analysis all state transition instructions are considered *interesting*, whereas for the linear constant analysis, all branch conditions are considered *interesting*, which is useful when eliminating dead code, for example. All jump functions at those interesting instructions are ignored by the garbage collector. We exclude the instruction-interaction analysis for RQ_2 as its post-processing needs the results at all instructions [21] rendering the garbage collection useless. To examine the influence of the jump functions garbage collector on the analysis, we compute the speedups of the **IDESolver++** compared to its corresponding versions without GC. We consider the configuration with the highest speedup to perform best.

6.3 Results

We have conducted our experiments on the 31 real-world C/C++ programs listed in Table 1. Although we have already argued on the correctness of our optimizations, we ran an additional, non-measured analysis batch to confirm that the new **IDESolver++** indeed computes the same results as the **IDESolver**. In what follows, we detail on the results of our experiments and answer the before defined research questions.

6.3.1 Baseline

Our evaluation of the baseline shows that in almost all measured configurations the **IDESolver++** is faster and consumes less memory than the **IDESolver**. We measured runtime speedups ranging from $1.16\times$ to $7.2\times$ on average and memory savings from $0.96\times$ to $4.8\times$ compared to the **IDESolver** as shown in Table 3. Due to the variance, the benefits of using our **IDESolver++** may be program dependent. Note that sometimes the **IDESolver++** consumes more memory in the typestate analysis than the **IDESolver**. This is because the **IDESolver++** allocates large buffers in advance to lower the number of re-allocations (cf. Section 5); in addition, the typestate analysis is very sparse; it propagates only a very small number of data-flow facts and therefore does not fill out the pre-allocated buffers which we do not consider as a problem since the total memory usage is negligible.

In contrast to the **IDESolver**, the **IDESolver++** ran out-of-memory very rarely, as is apparent in Figure 4. However, the figure also shows that the number of timeouts is higher for the **IDESolver++** than for the **IDESolver**. That is because analyses that ran out-of-memory in the **IDESolver** were able to run long enough to exceed the given time budget in the **IDESolver++**. All of the experiments that completed with the **IDESolver** were also completed with the **IDESolver++**, showing that the performance does not degrade. In fact, out of the 7 experiments that exceeded the time limit of four hours, three were solved in time with the new solver; out of the five experiments that ran out of memory, one can now be completed within the memory limit of 250GiB. Furthermore, all 7 experiments that required up to 143GiB of RAM can now be solved on an consumer hardware with only 32GiB RAM.

There are several aspects that contribute to the improvements in this baseline experiment. The most notable ones are: The elision of the multi-index storage for jump functions (see Section 4.1.2), the batch-processing (see Algorithm 1) of data-flow fact propagations, and the switch from the `std::unordered_map` to `l1vm::SmallDenseMap` (see Section 5).

Table 3 The average speedups/memory savings of the IDESolver++ with JF_{old} compared to PhASAR’s IDESolver together with their standard deviations

Analysis	Memory	Runtime
IIA	4.811 ± 1.192	7.227 ± 2.042
LCA	1.729 ± 0.365	4.683 ± 2.150
Typestate	0.968 ± 0.050	1.162 ± 0.143

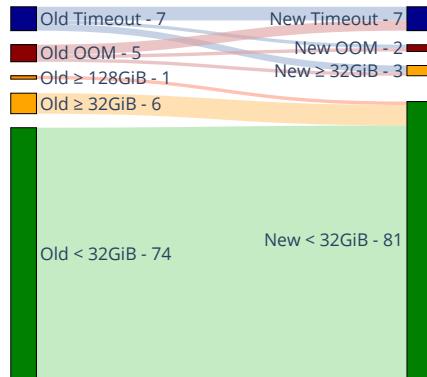


Figure 4 A sankey-plot showing how the number of (target program × analysis type) that finish with out-of-memory (OOM), timeout, or completed changes when switching from PhASAR’s IDESolver (Old) to our IDESolver++ (New) with JF_{old} keeping the time-limit of four hours and the memory limit of 250GiB.

Table 4 Results of our per-analysis comparison between the jump-function representations within our IDESolver++. We report the mean speedup and its standard deviation for both runtime and memory. Cells highlighted with green background indicate the JF with highest runtime speedup or memory savings for that analysis. In case, the highest speedup is <1 or the difference to the other jump-functions representations is not significant, we omit the highlight.

	JF1		JF2		JF3	
	Memory	Runtime	Memory	Runtime	Memory	Runtime
IIA	1.270 ± 0.231	0.927 ± 0.059	1.382 ± 0.230	0.949 ± 0.071	1.371 ± 0.221	0.957 ± 0.096
LCA	1.126 ± 0.097	0.939 ± 0.102	1.406 ± 0.267	1.064 ± 0.061	1.400 ± 0.261	1.063 ± 0.061
Typestate	1.059 ± 0.053	0.996 ± 0.023	1.057 ± 0.042	1.013 ± 0.035	1.057 ± 0.042	1.005 ± 0.022

Hence, we can already conclude that based on the high speedups for both runtime and memory as well as avoiding out-of-memory events, it is crucial to implement IDE in a performance-oriented way and just changing the implementation of the same underlying IDE algorithm can enable analyses that were not feasible before.

6.3.2 RQ₁: Jump-Functions Table Structure

We evaluated all three data structures JF_{ND}, JF_N, and JF_{NE}. We found that they behave differently depending on the target program and analysis. As expected, the instruction-interaction analysis puts a high load onto the solver, whereas the typestate analysis is very sparse and therefore completes within seconds.

Figure 5 shows both the runtime speedups and the memory savings of the different jump-functions representations compared to the deeply nested jump-functions representation JF_{old}. Both the runtime speedups and memory savings differ depending on the client analysis and have high variance over the target programs. In the (left) runtime speedup plot we can

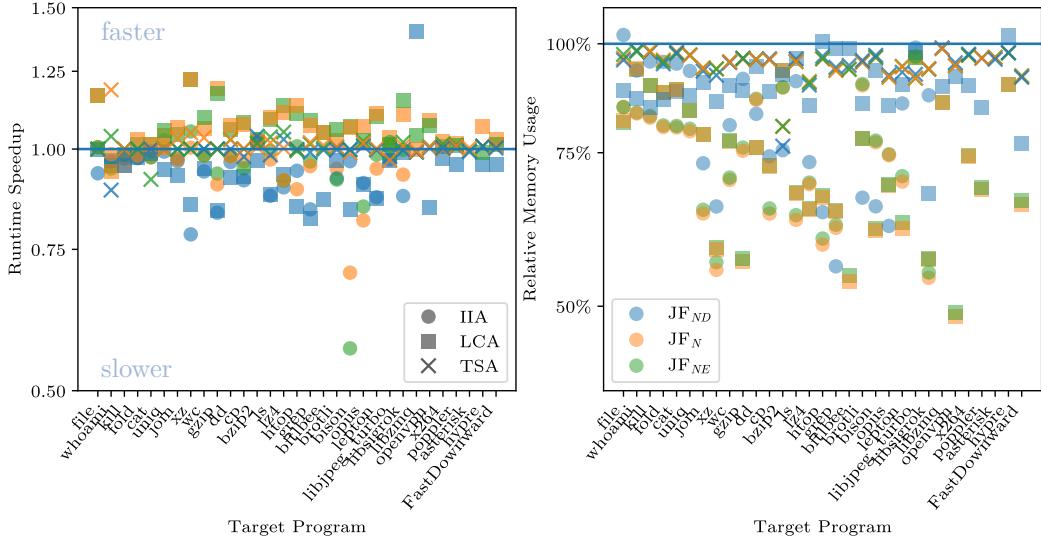


Figure 5 Scatter plots showing the IDESolver++ with the proposed jump-functions representations compared to the IDESolver++ using the nested representation inherited from PhASAR’s current IDESolver. The left plot shows the runtime speedup (higher is better), whereas the right plot shows the relative memory usage (smaller is better). The target programs are sorted in ascending order based on their number of LLVM-IR instructions. The IDESolver++ was configured to use JF_{ND} (blue), JF_N (orange), and JF_{NE} (green). The both horizontal lines are set at 1 meaning no speedup. We use a log-scale to account for the non-linear distribution of speedups.

see that the speedups of the analyses are approximately centered around 1 with a small advantage of JF_N and JF_{NE} over JF_{ND} for the LCA. In the (right) relative memory usage plot, it becomes visible that the IIA and LCA consume less memory with any of the proposed jump-functions representations than with JF_{old}. However, the variance across the analyzed target programs is high. For the TSA, the relative memory consumption is close to 94% for all jump-functions representations. The target programs in the plots of Figure 5 are sorted in ascending order by their number of LLVM-IR instructions. We provide variants of these plots with different program orderings on our supplementary website (see visualizations). Still, the orderings did not show observable correlations between the speedups and any of the tested program characteristics.

So, there is no clear overall “best” jump-functions table structure, and project- and analysis specific tradeoffs have to be made. However, by taking an analysis-centric view, we can determine the “best” jump-functions representation per analysis as shown in Table 4. For the IIA, JF_N has highest average memory improvement with 1.382×(consuming 72% of the memory from JF_{old}), but the significance test shows that the difference to JF_{ND} and JF_{NE} is not significant, so in terms of memory, they share the first place. In terms of running time, JF_{old} performed significantly best. For the LCA, JF_N is best in terms of both runtime and memory improvements, consuming only 71% of the memory from JF_{old} while being 6.4% faster; the difference to JF_{NE} is not significant, so we consider both JF_N and JF_{NE} best for the LCA. While for memory improvement, JF_{ND} is with using 97% of the memory slightly, but significantly better than JF_{old}, for runtime speedup, the difference between JF_{ND} and JF_{old} is insignificant. Finally, for the typestate analysis, the jump-functions representations performed similarly; yet the memory improvement of JF_{ND}, JF_N, and JF_{NE} over JF_{old} is significant, consuming around 94% of the memory from JF_{old}.

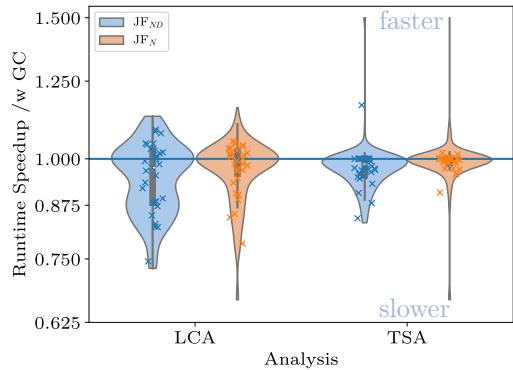


Figure 6a. A violin plot showing the runtime speedups of the IDESolver++ with garbage collection compared to their versions without GC. The solver was configured to use JF_{ND} (blue) and JF_N (orange). Note, that the y-axis is in log-scale to account for the non-linear distribution of speedups < 1 (slowdowns).

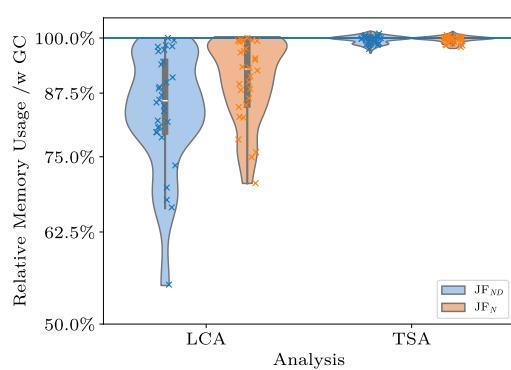


Figure 6b. A violin plot showing the relative memory usage of the IDESolver++ with GC compared to its versions without GC. The solver was configured to use JF_{ND} (blue) and JF_N (orange). We use a log-scale for the relative memory usages here.

■ **Figure 6** Violin plots showing the impact of enabling garbage collection on runtime and memory usage of the IDESolver++.

To answer *RQ₁*: The performance of the jump-functions representations highly depends on the performed analysis. However, JF_N and JF_{NE} have shown significantly best memory usage for the LCA and perform well for the IIA and TSA; this makes them a generally reasonable default choice. We also conclude that picking the right data structure oftentimes is *no* tradeoff between runtime speedup and memory savings; the same data structure can improve runtime and memory usage at the same time.

6.3.3 RQ₂: Jump-Functions Garbage Collection

The results of evaluating the jump functions garbage collector with JF_N are shown in Figure 6a and Figure 6b. For the LCA we see memory savings, where the analysis consumed, on average, 12% less memory ($\pm 10\%$). Furthermore, Figure 6b shows higher memory savings with JF_{ND} than with JF_N . For the TSA, the analysis with GC saved around 0.4% memory, which is significant, but we consider it negligible in most cases. This is expected because the TSA is very sparse and therefore does not have much to erase during garbage collection. Some analysis runs consumed even more memory than with disabled garbage collection. This is because of the additional book keeping meta-data that the garbage collector requires. In summary, the generalization to IDE indeed makes the GC less effective, but still it can drastically reduce the memory footprint of IDE analyses.

As expected, enabling jump functions garbage collection has non-negligible runtime-performance impact. The reason for this is that – in contrast to the experiments of Arzt [1] – the GC runs in the same thread as the analysis and therefore blocks the analysis process while performing the garbage collection. However, the mean speedup is close to 1 with 96.6% ($\pm 8.6\%$) for LCA and 98.3% ($\pm 6.3\%$) for TSA. Hence, the average runtime cost is still low.

Enabling the GC in single-threaded mode is a tradeoff between runtime and memory, as the GC reduces the memory consumption of IDE at the cost of increased runtime.

To answer *RQ₂*: Constraining the jump functions garbage collector to work in a single-threaded scenario results in a reduction of the memory consumption of the linear constant analysis of 12%, with only minimal runtime overhead. However, the effectiveness of the GC compared to the original GC from Arzt [1] is reduced, making it impractical for smaller analyses, and for those that do not propagate many data-flow facts.

6.4 Threads to Validity

Internal Validity

Runtime measurement on modern computing systems is a challenging task due to automatic clock boost and throttling as well as context switches enforced by the operating system. This makes reliable runtime measurements hard. We therefore ran our experiments three times and report averages to compensate for this noise. In addition, we ran each experiment in isolation on equivalent machines, ensuring that no other task is running in parallel. Our experiments each utilize only one thread to minimize the influence of the OS scheduler on the measurements.

We evaluated our experiments on a fixed set of target programs, on which we verified that the **IDESolver++** produces the same results as the **IDESolver**. We cannot rule out that there are programs where the solvers produce different results because of bugs in the implementations of either of them. To mitigate this risk, we performed our evaluation on a large set of real-world programs and configured the IDE solvers with three different client analysis problems.

External Validity

The performance of the analysis solvers may be different depending on the target program, that is, there may be programs that we did not benchmark where the analysis solvers behave differently. To mitigate this threat, we selected a diverse set of target programs from various domains and with different sizes and complexities. Furthermore, we configured the analysis solvers with three differently complex analysis problems to have greatest possible variation. This gives us for the first time a comprehensive study on a substantial number of real-world C/C++ programs.

6.5 Discussion

In Section 6, we presented the results of our evaluation, some of them require interpretation.

We have observed that JF_N in many cases has a lower memory consumption than JF_{ND} . This can be explained by the distribution of jump functions: For many analyses an extra experiment run with statistics instrumentation shows that the average size of the inner maps in JF_{ND} is < 4 , but still with a high number of total jump functions. Hence, JF_{ND} pays the memory overhead of a hash map for the majority of jump functions, whereas JF_N and JF_{NE} oftentimes store more than 1000 elements in their inner maps which can lead to more efficient use of the provided memory.

On the other hand, depending on the access patterns of the jump-functions table, JF_{ND} can lead to faster jump functions access. For the IIA, we see drastic performance benefits of JF_{ND} and JF_{old} compared to JF_N and JF_{NE} when analyzing BISON. This can be explained by the handling of aliasing in the IIA. All aliases of a data-flow fact are propagated individually

in the IIA. Therefore, for memory-indirection statements, such as `store a to b`, for all aliases of the stored pointer a all aliases of the target pointer b must be generated, which are independent from each other. This leads to the same jump functions to be accessed multiple times, which may be faster if the inner maps do not incur memory indirections because they are small enough for small-size optimization.

Combining the measurements from our baseline (cf. Figure 3) with our specific optimizations from Section 4, we achieve the following overall mean speedups in the IDESolver++ compared to PhASAR’s current IDESolver: Memory improvements of $6.9\times$ for IIA, and $2.7\times$ for LCA; runtime speedups of $6.9\times$ for IIA, and $4.9\times$ for LCA. For the typestate analysis, there is no overall mean speedup, but also no mean slowdown.

7 Related Work

Performance problems of IDE implementations are a known issue. He et al. [9] perform sparsification on the ESG by propagating data-flow facts not along ICFG edges, but on their corresponding def-use chains. Arzt and Bodden [3] automatically generate IDE summaries for libraries, which prevents re-analyzing commonly used libraries and lowers the size of the analyzed target programs. Arzt and Bodden [2] improve re-analysis of already analyzed programs by incrementally analyzing only the changes compared to the previously analyzed version. These approaches let any existing implementation of IDE scale better in the circumstances that they optimize. Nonetheless, they can still further profit from an improved solver that scales better in the first place.

Weiss et al. [29] use a database system to store their internal data structures partially on disk effectively increasing the amount of available memory. However, they focus on the specific problem of error-code propagation and do not generalize to arbitrary IDE analyses. Hsu et al. [10] propose a modified IFDS algorithm that no longer needs to store the ESG explicitly and computes the reachability based on Depth-First Tree Intervals instead. While this approach works well for IFDS problems, it cannot be applied to IDE problems directly as composing edge functions requires to store the jump functions in some way.

He et al. [8] improve the garbage collection presented by Arzt [1] by increasing the GC’s granularity from method-level to data-flow fact level. However, it suffers from the same restrictions of required multi-threading and also only applies to the same subset of IDE as the original garbage collector [1] that we generalize in this paper.

Apart from IDE, there are other approaches to precise interprocedural static data-flow analysis, such as weighted pushdown systems (WPDS) [12, 18]. As WPDS has the same runtime- and memory complexities as IDE, similar optimizations as the ones presented in this paper may be possible for WPDS as well. Other approaches, such as Boomerang [25] reduce their resource requirements by conducting demand-driven analyses, only computing the data-flow information for specific program locations. While demand-driven analyses work well for pointer analysis where a client analysis requests the demand, exhaustive taint analyses, e.g., a use-after-free analysis would need to issue a demand for each potential sink statement effectively degenerating the demand-driven analysis to a whole-program analysis with similar performance issues.

Yu et al. [31] tackle the performance problem by bringing data-flow analysis to the GPU and optimizing the algorithm, as well as the data-layout for GPU processing. As the CPU and GPU are particularly different hardware components, optimizations for GPU programs usually do not apply to CPU programs, and vice versa.

8 Conclusion

Current state-of-the-art IDE implementations do not scale well to large programs preventing the analysis of many interesting data-flow problems that can be used for bug- and vulnerability detection, as well as other important fields in software engineering. Based on years of experience with implementing and using IDE-based program analyses, we identified two different optimizations of the IDE algorithm. We found that choosing an efficient representation for the jump-functions table structure within the solver implementation has great influence on the performance of the algorithm. Still, it requires further research to select the right data structure for an analysis, or to even automate this process. Yet, we learned that an implementation of IDE has to be designed with performance in mind from the beginning to achieve a scalable implementation. Furthermore, we extended the jump functions garbage collection from Arzt to general IDE problems and removed the restriction to a multi-threaded solver implementation. We evaluated that it still reduces the memory footprint of the IDE analyses, though being less effective than the original.

Our experiments on 31 real-world C/C++ programs show runtime and memory speedups of up to $7\times$ on average compared to the existing IDE implementation in PhASAR and enable the analysis of more target programs than before. We found that especially extremely heavy analyses such as the instruction interaction analysis presented by Sattler et al. [21] can now be run on medium-to large programs that was not possible previously, even with larger server hardware. Still, some analyses require too much memory for being executed on an ordinary developer machine.

References

- 1 Steven Arzt. Sustainable Solving: Reducing The Memory Footprint of IFDS-Based Data Flow Analyses Using Intelligent Garbage Collection. In *Proc. Int. Conf. Software Engineering (ICSE)*, pages 1098–1110. IEEE, 2021.
- 2 Steven Arzt and Eric Bodden. Reviser: Efficiently Updating IDE-/IFDS-Based Data-Flow Analyses in Response to Incremental Program Changes. In *Proc. Int. Conf. Software Engineering (ICSE)*, pages 288–298. ACM, 2014.
- 3 Steven Arzt and Eric Bodden. StubDroid: Automatic Inference of Precise Data-Flow Summaries for the Android Framework. In *Proc. Int. Conf. Software Engineering (ICSE)*, pages 725–735. ACM, 2016.
- 4 Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Ochteau, and Patrick D. McDaniel. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, pages 259–269. ACM, 2014.
- 5 Eric Bodden. Inter-Procedural Data-Flow Analysis with IFDS/IDE and Soot. In *Proc. Int. Workshop on State Of the Art in Java Program Analysis (SOAP)*, pages 3–8. ACM, 2012.
- 6 Eric Bodden. The secret sauce in efficient and precise static analysis: the beauty of distributive, summary-based static analyses (and how to master them). In *Comp. Proc. ISSTA/ECOOP Workshops*, pages 85–93. ACM, 2018.
- 7 Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. Practical memory leak detection using guarded value-flow analysis. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, pages 480–491. ACM, 2007.
- 8 Dongjie He, Yujiang Gui, Yaoqing Gao, and Jingling Xue. Reducing the Memory Footprint of IFDS-Based Data-Flow Analyses using Fine-Grained Garbage Collection. In *Proc. Int. Symp. Software Testing and Analysis (ISSTA)*, pages 101–113. ACM, 2023.
- 9 Dongjie He, Haofeng Li, Lei Wang, Haining Meng, Hengjie Zheng, Jie Liu, Shuangwei Hu, Lian Li, and Jingling Xue. Performance-Boosting Sparsification of the IFDS Algorithm with Applications to Taint Analysis. In *Proc. Int. Conf. Automated Software Engineering (ASE)*, pages 267–279. IEEE, 2020.

- 10 Min-Yih Hsu, Felicitas Hetzelt, and Michael Franz. DFI: An Interprocedural Value-Flow Analysis Framework that Scales to Large Codebases. *Comput. Research Repository*, abs/2209.02638, 2022.
- 11 Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. Model checking for weakly consistent libraries. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, pages 96–110. ACM, 2019.
- 12 Akash Lal, Thomas Reps, and Gogul Balakrishnan. Extended Weighted Pushdown Systems. In *Proc. Int. Conf. Computer Aided Verification (CAV)*, pages 434–448. Springer-Verlag, 2005.
- 13 Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. Int. Symp. Code Generation and Optimization (CGO)*, pages 75–88. IEEE, 2004.
- 14 Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In Defense of Soundness: A Manifesto. *Commun. ACM*, 58(2):44–46, 2015.
- 15 Nomair A Naeem, Ondřej Lhoták, and Jonathan Rodriguez. Practical Extensions to the IFDS Algorithm. In *Proc. Int. Conf. on Compiler Construction (CC)*, pages 124–144. Springer-Verlag, 2010.
- 16 Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static detection of asymptotic performance bugs in collection traversals. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, pages 369–378. ACM, 2015.
- 17 Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proc. Symp. Principles of Programming Languages (POPL)*, pages 49–61. ACM, 1995.
- 18 Thomas Reps, Stefan Schwoon, and Somesh Jha. Weighted Pushdown Systems and Their Application to Interprocedural Dataflow Analysis. In *Proc. Int. Symp. Static Analysis (SAS)*, pages 189–213. Springer-Verlag, 2003.
- 19 Atanas Rountev, Mariana Sharp, and Guoqing Xu. IDE Dataflow Analysis in the Presence of Large Object-Oriented Libraries. In *Proc. Int. Conf. on Compiler Construction (CC)*, pages 53–68. Springer-Verlag, 2008.
- 20 Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theor. Comput. Sci.*, 167(1-2):131–170, 1996.
- 21 Florian Sattler, Sebastian Böhm, Philipp Dominik Schubert, Norbert Siegmund, and Sven Apel. SEAL: Integrating Program Analysis and Repository Mining. *ACM Trans. Softw. Eng. Methodol.*, 32(5):121:1–121:34, 2023.
- 22 Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. PhASAR: An Inter-procedural Static Analysis Framework for C/C++. In *Proc. Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 393–410. Springer-Verlag, 2019.
- 23 Philipp Dominik Schubert, Richard Leer, Ben Hermann, and Eric Bodden. Know your analysis: How instrumentation aids understanding static analysis. In *Proc. Int. Workshop on State Of the Art in Program Analysis (SOAP)*, pages 8–13. ACM, 2019.
- 24 M Sharir and A Pnueli. *Two approaches to interprocedural data flow analysis*. New York Univ. Comput. Sci. Dept., 1978.
- 25 Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 22:1–22:26. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016.
- 26 Yulei Sui, Ding Ye, and Jingling Xue. Static memory leak detection using full-sparse value-flow analysis. In *Proc. Int. Symp. Software Testing and Analysis (ISSTA)*, pages 254–264. ACM, 2012.
- 27 Yulei Sui, Ding Ye, and Jingling Xue. Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis. *IEEE Trans. Software Eng.*, 40(2):107–122, 2014.

- 28 Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. DangSan: Scalable Use-after-free Detection. In *Proc. Europ. Conf. Computer Systems (EuroSys)*, pages 405–419. ACM, 2017.
- 29 Cathrin Weiss, Cindy Rubio-González, and Ben Liblit. Database-backed program analysis for scalable error propagation. In *Proc. Int. Conf. Software Engineering (ICSE)*, pages 586–597. IEEE, 2015.
- 30 Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Spatio-temporal context reduction: a pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *Proc. Int. Conf. Software Engineering (ICSE)*, pages 327–337. ACM, 2018.
- 31 Xiaodong Yu, Fengguo Wei, Xinming Ou, Michela Becchi, Tekin Bicer, and Danfeng Daphne Yao. GPU-Based Static Data-Flow Analysis for Fast and Scalable Android App Vetting. In *Int. Symp. Parallel and Distributed Processing (IPDPS)*, pages 274–284. IEEE, 2020.
- 32 Zhiqiang Zuo, Yiyu Zhang, QiuHong Pan, Shenming Lu, Yue Li, Linzhang Wang, Xuandong Li, and Guoqing Harry Xu. Chianina: an evolving graph system for flow- and context-sensitive analyses of million lines of C code. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, pages 914–929. ACM, 2021.

Java Bytecode Normalization for Code Similarity Analysis

Stefan Schott  

Paderborn University, Germany

Serena Elisa Ponta  

SAP Security Research, Mougins, France

Wolfram Fischer  

SAP Security Research, Mougins, France

Jonas Klauke  

Paderborn University, Germany

Eric Bodden  

Paderborn University, Germany

Fraunhofer IEM, Paderborn, Germany

Abstract

Analyzing the similarity of two code fragments has many applications, including code clone, vulnerability and plagiarism detection. Most existing approaches for similarity analysis work on source code. However, in scenarios like plagiarism detection, copyright violation detection or Software Bill of Materials creation source code is often not available and thus similarity analysis has to be performed on binary formats. Java bytecode is a binary format executable by the Java Virtual Machine and obtained from the compilation of Java source code. Performing similarity detection on bytecode is challenging because different compilers can compile the same source code to syntactically vastly different bytecode.

In this work we assess to what extent one can nonetheless enable similarity detection by *bytecode normalization*, a procedure to transform Java bytecode into a representation that is identical for the same original source code, irrespective of the Java compiler and Java version used during compilation. Our manual study revealed 16 *classes of compilation differences* that various compilation environments may induce. Based on these findings, we implemented bytecode normalization in a tool jNORM. It uses Jimple as intermediate representation, applies common code optimizations and transforms all classes of compilation difference to a normalized form, thus achieving a representation of the bytecode that is identical despite different compilation environments.

Our evaluation, performed on more than 300 popular Java projects, shows that solely the act of incrementing a compiler version may cause differences in 46% of all resulting bytecode files. By applying bytecode normalization, one can remove more than 99% of these differences, thus acting as a crucial enabler for subsequent applications of bytecode similarity analysis.

2012 ACM Subject Classification Software and its engineering → Compilers

Keywords and phrases Bytecode, Java Compiler, Code Similarity Analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.37

Supplementary Material Software (ECOOP 2024 Artifact Evaluation approved artifact):
<https://doi.org/10.4230/DARTS.10.2.20>

Funding This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (GZ: SFB 901/3) under the project number 160364472.

 © Stefan Schott, Serena Elisa Ponta, Wolfram Fischer, Jonas Klauke, and Eric Bodden;
licensed under Creative Commons License CC-BY 4.0
38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 37; pp. 37:1–37:29



1 Introduction

In the past, researchers have developed many approaches for code similarity analysis on Java applications [50, 35, 52, 31, 60, 47]. These techniques target a wide variety of applications, like code clone detection, plagiarism detection, copyright infringement investigation, program comprehension, vulnerability detection and many more [49, 37]. Most developed techniques operate on source code. However, an application's source code is not always available, since applications are typically distributed in binary form. Especially in scenarios where external dependencies are included into a software product, often only the binary form is included without the corresponding source code. In case of Java, applications are distributed as JAR-archives that contain the *bytecode* of the application. Instead of compiling the source code directly to executable machine code, Java compilers generate an intermediate representation called *bytecode*, which is translated into machine code during execution time by the Just-in-time (JIT) compiler within the Java Virtual Machine (JVM). With the European Union's Cyber Resilience Act [11] coming into force soon and the US's Executive Order on Improving the Nation's Cybersecurity [6] already being effective, the creation of Software Bill of Materials (SBOM) has become mandatory. However, creating a faithful SBOM for Java applications is a difficult undertaking due to current tool's reliance on metadata [5, 13]. To reliably create such SBOMs an approach needs to be established that is able to find all used components based on the similarity of bytecode, since source code is generally not available.

There are only few approaches that have been developed for similarity analysis based on bytecode [4, 36, 58]. This may be due to the increased complexity when trying to compare bytecode instead of source code. As Dann et al. [12] and Kononenko et al. [38] have shown, the comparison of bytecode is more complex than the comparison of source code, since equal source code is compiled into different bytecode, depending on the compiler, version and configuration used. While the generated bytecode is semantically equivalent, its syntactic structure may vastly differ. To overcome this difficulty we investigate the utility of *bytecode normalization* to create a representation that is *independent* of the environment that has been used for compilation. This independent representation can subsequently be used by bytecode-based code clone, plagiarism or vulnerability detectors without the need for consideration of compilation environments, significantly simplifying their task. Our approach to achieve bytecode normalization, which builds upon Dann et al.'s approach [12], is a procedure that

1. translates the bytecode into Jimple, the primary intermediate representation of the Soot [55] bytecode optimization framework, which reduces the more than 200 available bytecode-instructions to only 15 different Jimple instructions,
2. as a baseline first applies common optimizations like constant propagation, dead code removal and unconditional branch folding to further reduce differences, and then specifically, and lastly
3. transforms *compilation differences* induced by different compilation environments.

We uncovered the set of compilation differences by systematically comparing bytecode of popular Java libraries generated by different vendors, versions and configurations of Oracle's Java Development Kit's (JDK) and OpenJDK's compiler (javac). During this initial study, we found a total of 16 classes of compilation differences.

We implemented bytecode normalization in a tool JNORM, and evaluated it on more than 300 of the most popular Java projects on GitHub by compiling the same source code within various compilation environments with different compiler vendors, versions and target levels. The evaluation shows that even a single increase of the compiler version may result in up to 46% of all generated bytecode files containing compilation differences. By applying JNORM's

bytecode normalization one can reduce these differences by more than 99%. Thus, bytecode normalization can function as an important enabler for bytecode similarity analysis in all cases in which source code is not available.

To summarize, this paper makes the following original contributions:

- It investigates the usage of different Java compilers and settings in real-world projects.
- It presents a comprehensive set of 16 classes of compilation differences that are induced when using different vendors, versions or target level configurations of the JDK's and OpenJDK's Java compiler.
- It presents an approach to bytecode normalization, implemented in a tool `JNORM`, to virtually completely remove the differences introduced by compiling the same source code within different compilation environments.
- It evaluates `JNORM` on a large set of real-world Java applications collected from GitHub.

The remainder of this paper is structured as following. Section 2 introduces terms and concepts related to similarity analysis, Java compilation and normalization. Afterwards, Section 3 presents the concept of Java bytecode normalization implemented in `JNORM` and an overview of the uncovered compilation difference classes. Section 4 presents an evaluation of `JNORM` on a set of real-world Java projects. Related work is presented in Section 5. We discuss possible threats to validity in Section 6 and conclude in Section 7.

`JNORM`, its source code, more detailed evaluation results and a study on the usage of Java compilers and target levels are publicly available at:

<https://doi.org/10.5281/zenodo.12625104>

2 Background

This section introduces concepts that are related to code similarity analysis and the compilation of Java applications.

2.1 Code Similarity Analysis

Code similarity analysis is a technique that seeks to determine the similarity of two or more code fragments. The calculation of the similarity of code fragments has a large number of uses, like code clone, plagiarism, licensing violation, malware or vulnerability detection [49, 37].

Depending on the desired application area different techniques are employed. Text- [50, 4] or token-based [35, 52] techniques try to find similarities within the textual information of the code fragments. Tree-based techniques [31] try to additionally leverage syntactic information of the code for the similarity analysis. Some techniques even try to find semantic similarities within code fragments [39]. These techniques are typically graph-based and offer low potential for scalability [49]. Recently machine learning based approaches [60, 51], which typically train a classifier that decides how similar code fragments are, have become popular. In terms of efficiency and scalability, text- and token-based techniques, which can solely focus on syntactic features, are much preferred.

Typically the similarity analysis is performed on a source code level. However, the source code of a compiled binary is not always available or trustworthy. In such scenarios, e.g. plagiarism detection or SBOM generation, the similarity analysis has to be performed on the binary itself. However, the resulting binary code is highly dependent on the environment it has been compiled in, i.e. different compiler versions or settings produce different code, even when compiling the same source code [12]. This characteristic makes binary similarity analysis a much more complex task.

2.2 Java compilers

In contrast to other compiled programming languages like C or Go, Java applications are not directly compiled into machine-executable code, but into Java *bytecode*. This bytecode is executed by the Java virtual machine (JVM), which comes with a just-in-time (JIT) compiler that compiles the bytecode into executable machine code at runtime. Because of this, the Java bytecode has the following characteristics:

1. **Platform independence:** The JVM architecture aims at platform independence. The generated bytecode is independent from the platform it is intended to run on [46].
2. **Unoptimized bytecode:** Optimizations are performed during runtime by the JIT compiler, therefore compiled bytecode is typically not optimized [24].

Because of these characteristics, the amount of variance across generated bytecode is not as high, when compared to machine-code, since there are no different optimization levels or differences due to the targeted platform.

There exist multiple different Java compilers like e.g. Oracle’s JDK or OpenJDK’s compiler [45], Eclipse’s JDT core compiler [28], IBM’s Jikes compiler [32], or the GNU Compiler for Java (GCJ) [19]. The JDK’s and OpenJDK’s compilers can be invoked programmatically or through the command-line application *javac* that comes pre-shipped with each JDK. Given the same source code as input, in many cases these compilers produce *different* Java bytecode.

In general, Java compilers support source codes that adhere to different versions of the language specification and can generate bytecode for different JVM versions lower than the compiler’s version. This backwards compatibility can be used by setting the compiler’s *target level*. For example, bytecode compiled with a JDK11 compiler with target level set to 8 can be executed by a JVM only supporting up to Java 8. The set of available target levels for a compiler is usually limited to a subset of earlier versions.

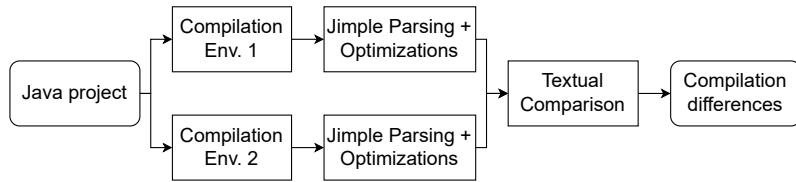
2.3 Jimple

Jimple is an intermediate representation (IR) of Java bytecode that was designed for providing a format that allows for simplified analysis, optimization and code transformations. Jimple maps the more than 200 Java bytecode instructions to only 15 different Jimple instructions in a *three-address* based representation. Three-address based representation means, that each instruction generally contains at most three different operands, e.g., one used for the left-hand side of an assignment and two used for binary operations on the right-hand side. This restriction greatly simplifies the processing of individual IR statements, which is why three-address IRs are nowadays commonplace. During the transformation Jimple retains all the type information present in the bytecode.

Jimple is the primary IR of the most popular Java bytecode optimization and analysis framework SOOT [55]. Alongside various code optimization options, SOOT provides an API to conveniently transform Jimple instructions. SOOT can automatically convert Java bytecode to Jimple (and vice-versa).

3 Java Bytecode Normalization

As we show next, Java bytecode normalization allows for the removal of differences in Java bytecode that are solely introduced by the usage of different compilation environments. In the following we describe how we detected the compilation differences in the first place, as well as the details of our bytecode normalization approach and its implementation in jNORM.



■ **Figure 1** Setup to determine compilation differences.

3.1 Investigation of Compilation Differences

Before the development of our bytecode normalization approach jNORM, we performed a study to investigate the differences induced by different compilation environments. Figure 1 shows the setup we used to determine compilation differences. For each comparison, we supplied the source code of various versions of the popular Java libraries Apache commons-io, Apache commons-lang, Jackson-databind, SLF4J and Google Guava, to two different environments for compilation. Afterwards, we converted the resulting bytecodes to Jimple and applied code optimizations, provided by the SOOT framework, to reduce dissimilarities. Finally, we performed a textual comparison on the optimized Jimple representations to determine the remaining compilation differences. Two files were considered different, and manually inspected by the authors, as soon as one character differed in the textual comparison.

Our compilation environments included the javac compilers shipped with JDKs 5–8, 11, and 17. Moreover, this version-range covers all Long-Term-Support (LTS) versions of the Java ecosystem until August 2023. A usage study of Java compilers and target levels in Java projects, which revealed these to be the by far most relevant compilers and versions, is available within an electronic appendix in our provided artifact.

We consider three types of parameters, JDK vendor, JDK version, and Java target level. We used Oracle’s JDK, as well as OpenJDKs distributed by Amazon Corretto and Eclipse Adoptium.

In total, our setup revealed 16 compilation difference classes, present in the investigated projects, which are listed in Tables 1a and 1b. Table 1a shows the difference classes produced by changing the JDK version, while Table 1b shows the difference classes produced when adjusting the Java target level. We did not find any vendor-related difference classes in our initial experiments.

Furthermore, we inspected the official JDK release notes [27] related to newly released compiler versions. However, this inspection did not reveal any so far uncovered difference classes. Our evaluation performed on more than 300 of the most popular Java projects (see Sections 4.3 and 4.4) also revealed no additional difference classes.

In the following we describe jNORM’s approach to bytecode normalization and how it transforms the identified compilation difference classes into a representation that is common across all investigated compilation environments.

3.2 Overview of jNorm

Figure 2 depicts an overview of jNORM. First, jNORM parses a Java bytecode file (.class file) into Jimple format, which is specifically designed for efficient optimizations and transformations. Note that jNORM also has the capability to process multiple bytecode files at once, and therefore full Java projects, but because each file is normalized independently of others we will explain bytecode normalization of single files. To reduce the initial set of differences for the following steps of the normalization process, jNORM applies different types of common

Table 1 Difference classes on JDK version and Target level change.

(a) JDK version change.

ID	JDK	Compilation Difference Class
N1	5 → 6 & 7 → 8	Synthetically generated methods
N2	5 → 6	Arithmetic
N3	6 → 7	CharSequence toString invocation
N4	7 → 8	Empty try-catch-finally block
N5	7 → 8	String constant concatenation
N6	8 → 11	Method reference operator
N7	8 → 11	Buffer method invocation
N8	8 → 11	Try-with-resources
N9	8 → 11	Duplicate checkcasts
N10	11 → 17	Enums

(b) Target level change.

ID	Target	Compilation Difference Class
N11	6 → 7	Outer class object creation
N12	8 → 11	Dynamic string concatenation
N13	8 → 11	Nest-based access control
N14	8 → 11	Invocation of private methods
N15	8 → 11	Inner class instantiation
N16	multiple ^a	Insertion or removal of typechecks

^a This compilation difference class occurs across multiple JDK and target level changes.

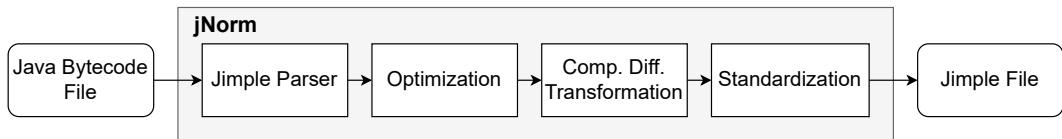


Figure 2 Overview of jNORM.

optimizations to the Jimple representation of the input bytecode file. Afterwards, in the Compilation Difference Transformation step (see Figure 2), jNORM handles the remaining set of compilation differences by performing certain transformations on the optimized Jimple representation. These transformations are targeted towards specific constructs that we found to be compiled differently based on the used compilation environment. jNORM detects these constructs within the target program and transforms them into a normalized representation. The applied transformations interfere with the naming scheme of local variables inside the target programs, which cause the introduction of new dissimilarities. jNORM handles these dissimilarities by standardizing (see Figure 2) the order and naming scheme of variables. After the normalization process is finished, jNORM outputs the normalized Jimple representation.

3.3 Jimple Parsing and Optimization

The first step of Java bytecode normalization consists of parsing the targeted bytecode file into a Jimple representation. This allows for a convenient application of common program optimizations provided by SOOT. We apply the following optimizations to each method [54]:

- **Copy Propagation:** Usages of variables in statements are replaced by their values, e.g. in a statement like `x = y + 3`, the reference to variable `y` is replaced by the value stored in `y`.
- **Constant Propagation and Folding:** Expressions that entirely consist of compile-time constants (e.g. `2 * 3`) are replaced by the constant result.
- **Dead Assignment Elimination:** Assignment statements to local variables, whose value is not subsequently used, are removed.
- **Conditional Branch Folding:** The expressions inside if-conditions are statically evaluated. If the expressions evaluate to constants, the unreachable conditional branch statements are removed.

- **Unconditional Branch Folding:** Unnecessary `goto` statements are removed.
 - **Unreachable Code Elimination:** Unreachable code is removed.
 - **Null Check Elimination:** Null-check statements, where the checked variable is known not to be null, are removed.
 - **Unused Local Elimination:** Unused local variables within a method are removed.
- These optimizations already contribute to a decrease of dissimilarities introduced during compilation [12]. However, after applying the optimizations, many important compilation differences still remain, which are targeted in the next step.

3.4 Compilation Difference Transformation

Through our investigation (see Section 3.1) we identified 16 compilation difference classes summarized in Tables 1a and 1b. In the following we describe the identified classes and the transformations applied by JNORM in detail.

The transformations that JNORM performs are not arbitrarily chosen. Each transformation produces a version that is generated by at least one compiler within our dataset. Furthermore, the decision whether to transform a compilation difference class to the older or the newer version is also not arbitrary. Typically one of the two versions contains more information than the other (e.g. a more specific return type in the newer version or the amount of string concatenation calls before their combination into a single call). As we cannot simply add information that is unavailable when only having access to the bytecode, we have to transform the difference class to the version that contains less information, therefore stripping some information from the generated bytecode. However, this information cannot be used for similarity analysis, since, based on the used compilation environment, it is not guaranteed to be present in the bytecode.

Note that we do not aim at generating an executable version of the bytecode with all semantics preserved, but at preserving information that is possibly important for a similarity analysis. Similarity analysis approaches that additionally require an executable version of the analyzed application, can use the original bytecode that has not been normalized, in addition to the normalized version.

N1: Synthetically generated methods

In many cases the JDK compiler synthetically generates methods into classes. Often this is used to generate bridge-methods that enable access to private members. Such synthetically generated methods are marked by the compiler with a specific `synthetic` flag [34]. Depending on the used JDK, these methods are not always generated in certain cases, e.g. whenever a method of a class uses a `Comparator` to create a specific ordering of objects, starting from JDK6 the compiler automatically generates a corresponding `sort` method into the class. Thus, such synthetic methods introduce differences and cannot be reliably used for a code comparison.

Transformation: JNORM removes such synthetic methods from the Jimple representation, as they cannot be modified within the source code anyway.

N2: Arithmetic

In some cases, integer subtractions are replaced with additions of negative numbers inside the bytecode produced by JDK6 and higher. A statement like `i1 = i1 - 5`, generated by JDK5, is replaced by a conversion of the positive number to a negative one (`i1 = (int) -5`) and a subsequent addition with the negative number like `i2 = i2 + i1`, by JDK6 and higher.

■ Listing 1 `toString()` invocation (Jimple)

```

1 java.lang.CharSequence r1;
2 java.lang.String r2;
3
4 // JDK6:
5 r2 = virtualinvoke r1.<java.lang.Object: java.lang.String toString()>();
6
7 // JDK7:
8 r2 = interfaceinvoke r1.<java.lang.CharSequence: java.lang.String toString()>();

```

Transformation: Whenever JNORM identifies an addition involving negative integers, it converts it into a subtraction.

N3: CharSequence `toString` invocation

The JDK7 compiler changed the way the `toString` method is handled in the bytecode when invoked on an object of type `CharSequence`. As it can be observed in Listing 1 (subtle differences are highlighted within the listings), the invoke type `interfaceinvoke` replaced `virtualinvoke`, and the more specific type `java.lang.CharSequence` replaced the method return type `java.lang.Object`.

Transformation: Whenever JNORM identifies a call to a `toString` method with a `java.lang.CharSequence` return type, it converts the method call to its previous, more generic, version.

N4: Empty try-catch-finally block

In most cases a try-catch-finally block comes with one or more catch blocks that react to some types of thrown exceptions. However, catch blocks can be empty or even missing completely. A try-catch-finally block with empty (or even missing) catch blocks is a syntactically valid Java construct, used to execute some instructions, no matter what happens in the try block. Prior to JDK8, the JDK compiler produces a redundant exception catching block¹ in the bytecode, if a catch block is empty or missing.

Transformation: If JNORM identifies such redundant exception catching blocks, it removes them from the Jimple representation of the bytecode.

N5: String constant concatenation

When using the JDK8 or higher compiler, string concatenation optimizations are introduced. Whenever multiple string constants are concatenated, compilers prior to JDK8 would use multiple calls to the `StringBuilder.append` method. A simple concatenation like

```
String helloWorld = "Hello " + "World!";
```

would result in two calls to the `StringBuilder.append` method, one receiving “Hello” and the other receiving “World!” as argument. However, as of JDK8, the compiler concatenates these two strings at compile time and produces a single call to `StringBuilder.append`. This holds true only for subsequent string constants: whenever a substring assigned to a variable is involved in the concatenation, multiple `StringBuilder.append` calls are used.

¹ In bytecode and Jimple there exists no notion of catch blocks. We use this terminology in synonym with exception traps.

■ **Listing 2** Method reference operator usage (Jimple)

```

1 org.apache.commons.io.IOFileFilter r0;
2
3 // JDK8:
4 virtualinvoke r0.<java.lang.Object: java.lang.Class getClass()>();
5
6 // JDK11:
7 staticinvoke <java.util.Objects:
8     java.lang.Object requireNonNull(java.lang.Object)>(r0);

```

■ **Listing 3** Buffer method invocation (Jimple)

```

1 java.nio.ByteBuffer r0;
2
3 // JDK8:
4 virtualinvoke r0.<java.nio.ByteBuffer: java.nio.Buffer flip()>();
5
6 // JDK11:
7 virtualinvoke r0.<java.nio.ByteBuffer: java.nio.ByteBuffer flip()>();

```

Transformation: When jNORM identifies subsequent calls to the `StringBuilder.append` method with string constants as arguments that are not referenced by variables, it combines them into a single call.

N6: Method reference operator

With the release of JDK8, the method reference operator (`::`) was introduced to the Java programming language. It allows one to refer to a method with the help of its declaring class or object name and is especially useful in combination with streams. Listing 2 shows how the operator usage is handled during compilation. Before performing the actual method call, if the operator is referring to a method of an object, a null check is performed at runtime. This is done to ensure that the object, the referred method belongs to, actually exists and is not `null`. The usual way to perform null checks in JDK8 and lower is to call the method `getClass` on the object to check. This mechanism was replaced in newer JDKs by invoking the static `requireNonNull` method.

Transformation: For normalization, jNORM transforms all occurrences back to the old null-checking mechanism.

N7: Buffer method invocation

Starting from JDK11, the return type of all subclasses of `java.nio.Buffer` was further specified. Instead of returning the type `java.nio.Buffer` (cf. Listing 3), newer JDKs further specify the return type. Listing 3 shows that methods of the class `java.nio.ByteBuffer`, compiled with JDK11, return `ByteBuffer` instead of `Buffer`. This holds true for every subclass of `java.nio.Buffer` and any method returning a `Buffer` object.

Transformation: When jNORM finds the invocation of a method of a `java.nio.Buffer` subclass with `Buffer` as return type, it transforms the return type to the more specific type.

N8: Try-with-resources

The try-with-resources statement allows to declare resources that are used within the statement, which are guaranteed to be closed at the end, no matter if an exception is thrown.

Whenever a try-with-resources statement is used in the source code, the JDK compiler produces multiple exception handlers that wrap each other in the bytecode, since the bytecode does not provide a separate instruction for such a statement. In some cases, prior to JDK11, these wrapped exception handlers are redundant, since they do not cover any application code but only automatically generated exception handling code. These redundant exception handlers are not created as of JDK11.

Transformation: Whenever jNORM identifies an exception handler that only covers automatically generated exception handling code, it removes the exception handler and its corresponding code from the declaring function.

N9: Duplicate checkcasts

Due to a bug [18] fixed in JDK11, earlier JDK compilers may insert the same `checkcast` instruction twice, one after the other.

Transformation: jNORM removes redundant typechecks for normalization, if it identifies such duplicates.

N10: Enums

Enums in Java are special types that can only take on certain predefined values. When an enum is created, the JDK compiler creates a separate class for each enum and defines the possible values inside the `clinit` function, which acts as a static initializer. In contrast to a constructor, which is called when an object of a class is initialized, the `clinit` function is called when the class itself is initialized. Prior to JDK17, the initialization of the possible enum values is performed directly inside the `clinit` method, while in JDK17 the definition is moved to its own function, which is called from `clinit`.

Transformation: If jNORM detects that the enum values are initialized within the `clinit` method, it moves the initializations into its separate method and calls this method from `clinit`.

N11: Outer class object creation

Changing the target level from Java 6 to Java 7 changes the generated bytecode, when an inner class creates an object of another sibling inner class within their shared outer class as shown in the following listing:

```
SiblingInnerClass sic = getOuterClass().new SiblingInnerClass();
```

In this case the method `getOuterClass` returns a reference to the outer class shared by both inner classes, the one that contains the above statement and the one that is created by the statement. Whenever this is the case, the compiler inserts a check to verify, that the method `getOuterClass` does not return `null`. This is done in the same way, as described for difference class N6, where the previous way of performing a null-check via the `getClass` method is replaced by a call to the `requireNonNull` method.

Transformation: jNORM transforms all occurrences back to the old null-checking mechanism, as it does for difference class N6.

■ Listing 4 String concatenation (Jimple)

```

1 int i0;
2 java.lang.StringBuilder $r0, $r1, $r2, $r3;
3 java.lang.String[] r5;
4
5 // Target Level 8:
6 $r0 = new java.lang.StringBuilder;
7 specialinvoke $r0.<StringBuilder: void <init>()>();
8 $r1 = virtualinvoke $r0.<StringBuilder:
9     StringBuilder append(java.lang.String)>("Amount: ");
10 $r2 = virtualinvoke $r1.<StringBuilder:
11     StringBuilder append(int)>(i0);
12 $r3 = virtualinvoke $r2.<StringBuilder:
13     StringBuilder append(java.lang.String)>(" Pieces");
14 virtualinvoke $r3.<StringBuilder: java.lang.String toString()>();
15
16 // Target Level 11:
17 dynamicinvoke "makeConcatWithConstants" <java.lang.String (int)>(i0)
18     <java.lang.invoke.StringConcatFactory:
19         java.lang.invoke.CallSite makeConcatWithConstants(
20             java.lang.invoke.MethodHandles$Lookup,
21             java.lang.String, java.lang.invoke.MethodType,
22             java.lang.String, java.lang.Object[]
23         )>("Amount: \u0001 Pieces");

```

N12: Dynamic string concatenation

In Java 11 and higher the old string concatenation approach of repeatedly calling the `StringBuilder.append` method (see N5), is replaced by a single `invokedynamic` instruction, which defers the resolution of a method call to runtime. This change was introduced to optimize the performance of string concatenations [25]. Listing 4 showcases the differences of string concatenation compiled for target levels 8 and 11. Previously, for each part of the string concatenation, one call of the `StringBuilder.append` method was required. However, in the new version, a dynamic approach that looks similar to template-based string building is generated. A single dynamic call of the `makeConcatWithConstants` method is performed, where string constants are concatenated into a single constant, while dynamic values are expressed by placeholders (see `\u0001` in line 23 in Listing 4) which are replaced by the resolved value during runtime.

Transformation: JNORM transforms the old string concatenation procedure into a template-based concatenation using `invokedynamic`.

N13: Nest-based access control

With the release of Java 11, a new concept for accessing members of inner classes, called nest-based access control [43], was introduced to the language specification. When inner classes are defined within a class, the JDK compiler compiles each inner class into its own file. The JVM treats each class as a separate entity and therefore disallows access to private members from methods outside of the class. However, the Java language specification *does* allow such access to private members of inner classes if they are originating *from the outer class* and vice versa. Prior to the release of Java 11, such access was handled by the compiler generating public bridge methods in the inner class for each private member, that the outer class can use to circumvent calling a private method. Starting from Java version 11, this

indirect access via generated bridge methods is not necessary anymore. A new property has been introduced that marks inner classes as *nestmates* of their outer class, which tells the JVM that access to private members is explicitly allowed between the marked classes. The JVM then automatically puts appropriate access-control checks into place. This change was introduced due to transparency, simplicity and security reasons.

Transformation: If jNORM finds classes that use bridge-methods to access private members of their respective inner classes, it transforms them to the nest-based access pattern created when specifying target level 11.

N14: Invocation of private methods

On top of adding nest-based access control, Java 11 comes with a new way to invoke private methods, even within the same class. Prior to Java 11, all private methods were invoked via the `invokespecial` instruction. With Java 11, to be consistent with the rules of the nest-based access control specification, certain private-method invocations were changed to use `invokevirtual` instructions [43].

Transformation: jNORM transforms private method invocations to use `invokevirtual` instead of `invokespecial`.

N15: Inner class instantiation

Going from Java 8 to Java 11, the instantiation of inner classes was changed. In some cases, in Java 8 and earlier, when an inner class is instantiated within the outer class, the JDK compiler generates an additional anonymous class that is empty. This behavior serves no apparent purpose and was removed in Java 11.

Transformation: If jNORM finds empty anonymous classes, it removes them.

N16: Insertion or removal of typechecks (aggressive transformation)

To check the type of an object, the bytecode instruction `checkcast` is used. Among other things, it is used when the developer performs a typecast on an object, so that the JVM can verify whether the specified type is suitable for the object. However, when changing the JDK version or target level, the compiler's behavior regarding typechecks changes. In contrast to the other compilation difference classes, this difference class cannot be isolated to a single version change, as it happens to different extents at various JDK version or target level changes. In some cases the compiler inserts `checkcast` instructions even though the developer did not write a typecast, or it does not place a `checkcast` instruction for typecasts placed by the developer. Whether the compiler places a `checkcast` instruction or not often depends on the used compilation environment.

Transformation: By default jNORM does not transform such typechecks, as we were not able to detect a pattern that indicates whether a typecheck should be removed or inserted, by just having access to the bytecode. Still, jNORM offers an *aggressive normalization mode* where it removes all `checkcast` instructions from the normalized Jimple representation of the bytecode. Such transformation removes information that can be used for similarity analysis and possibly changes the application's semantics rather than just adopting a format produced in a different compilation environment. In some cases, e.g. when the change between two bytecode fragments only consists of typecheck insertions or removals, this loss of information makes the normalized fragments indistinguishable.

<pre> 1 i1 = i1 - 1; 2 i2 = i1 + 10; 3 i3 = i2 + 10; </pre>	<pre> 1 i1 = (int) -1; 2 i2 = i1 + i1; 3 i3 = i2 + 10; </pre>	<pre> 1 i1 = i1 - 1; 2 i3 = i1 + 10; </pre>
(a) JDK5 (not normalized)	(b) JDK6 (not normalized)	(c) JDK6 (normalized)

Figure 3 Application of standardization (Jimple).

We leave a more thorough investigation of the patterns that indicate typecheck placements in the bytecode as future work.

3.5 Standardization

Since the names of local variables are removed by default after compiling Java source code into bytecode (bytecode uses an operand stack instead of local variables), all local variables within the Jimple representation are named by concatenating their inferred type with an ascending integer number. After applying transformations that create, remove, or reorder local variables, such as the Arithmetic or Try-with-resources transformations, the ordering of local variable definitions and their naming scheme might become inconsistent. Because of this, we remove unused local variables and reorder definitions of used local variables based on their usage order, which stays consistent during all optimizations and transformations. Afterwards, we rename the local variables based on their types and usage order. This ensures a standardized naming scheme across all methods, even after applying transformations.

Figure 3 shows why standardization is necessary in some cases. Listing 3a shows subtraction generated by the JDK5 compiler, while Listing 3b shows subtraction output by the JDK6 compiler. After applying normalization to the code fragment in Listing 3b (see N2: Arithmetic), we obtain the code shown in Listing 3c. Since we removed the intermediate variable `i2`, the logical naming following the variable deletion does not match up anymore to the version that did not require any normalization. Therefore we need to apply standardization and rename every following variable usage, to achieve a representation that is equal to the code fragment that did not require normalization.

4 Evaluation

In the following we evaluate JNORM’s normalization performance. To do so, we answer the following research questions.

RQ1: Does the JDK vendor influence the bytecode generation?

RQ2: How does JNORM perform on changing JDK versions?

RQ3: How does JNORM perform on changing Java target levels?

RQ4: To which degree can bytecode normalization support similarity analysis tools?

RQ5: How prevalent are the individual compilation difference transformations of JNORM?

The first three research questions focus on JNORM’s normalization performance within different compilation environments. Research question 4 investigates to which extent JNORM can support similarity analysis tools. The final research question gives an overview about the most common compilation difference classes. We used similar experimental setups for each of the research questions.

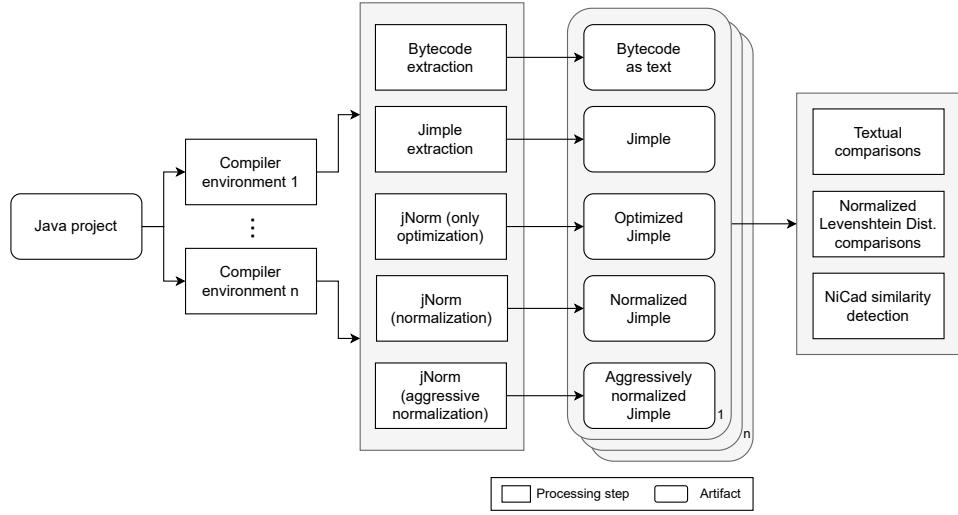


Figure 4 Overview of our experimental setup.

4.1 Experimental Setup

To evaluate JNORM’s normalization performance, we use the approach depicted in Figure 4.

We selected real-world Java projects based on the following process: At first, we used the GitHub search API to obtain the 1,000 projects with the most stars that have Java listed as their main language as of August 2023. We excluded two projects that, alongside Java files, also contained other JVM-based programming languages like Groovy or Clojure, as the compiled classes would interfere with further evaluation steps. Then we filtered out every project that does not use Maven as build tool, as Maven’s static configuration files in XML format, unlike Gradle, allow for an automated change of the compilation setup without knowing the project’s build structure in detail. After this step we were left with 322 Maven projects. Finally, we excluded two projects that, when compiled twice within the same environment, would produce different results, because of code generation at compilation time. This is typically due to files being generated for testing purposes or due to parser code being generated from a grammar, which in some cases produces random identifiers. This left us with a set of 320 Java projects, including tutorial projects, popular libraries, frameworks and real-world applications.

We cloned each project’s git repository. As automatic compilation is a known problem for Java projects [22], to increase the chances of a successful compilation in the next step, we then moved to the latest release tag (if available). As depicted in Figure 4, we compiled each of the projects within different compilation environments. We chose the compilation environments based on the setting we were interested in for the respective research questions.

To evaluate the normalization performance of JNORM, we applied different procedures to the compiled projects, as shown in Figure 4. The “Bytecode extraction” component in the figure uses ASM 9.3 [3] to extract the textual representation of the bytecode from the compiled class files (omitting all debug information). Furthermore, we additionally extracted the plain Jimple representation of the compiled classes in textual form without applying any optimizations or transformations. The plain bytecode and Jimple can be used as a baseline to establish the amount of differences induced by different compilation environments. To establish how the different normalization steps of JNORM contribute to the removal of compilation differences, we let JNORM run in different modes. “JNORM (only optimization)” only applies the Jimple parsing and optimizations described in Section 3.3. “JNORM (normalization)” applies all the steps described in Section 3 with transformations

■ **Table 2** JDKs considered in our evaluation.

JDK Version	Oracle JDK	AC OpenJDK	EA OpenJDK
7	1.7.0_80	—	—
8	1.8.0_333	8.342.07.4	8u352-b08
11	11.0.16	11.0.16.9.1	11.0.17+8
17	17.0.4.1	17.0.5.8.1	17.0.5+8

N1–N15, but keeps all typechecks in place (default normalization mode). “JNORM (aggressive normalization)” differs from the previous as it also removes all typechecks from the resulting Jimple representation. Applying all procedures, we obtain five sets of files per compilation environment and project:

- Extracted bytecode as text
- Extracted Jimple
- Optimized Jimple
- Normalized Jimple
- Aggressively normalized Jimple

We apply different comparisons to each of the resulting file sets generated within different compilation environments, resulting in multiple comparisons per project. At first we perform a textual head-to-head comparison on the file-, as well as method-level. To do so we compare files with the same fully qualified name and methods with the same signature to each other that were produced within different compilation environments. As soon as there is a single textual difference between the compared files or methods, they are classified as being *different*. If a method is present in one file, but not the other, it is classified as *disjunct*.

In addition to textual head-to-head comparisons, which only allow for a yes/no detection of equality, we calculate the normalized Levenshtein Distance (NLD) [59] between the compared files and methods. The NLD is a measure that is used to calculate the similarity of two text sequences. The Levenshtein Distance counts the number of required character insertions, deletions or substitutions to transform one text sequence into the other. The normalized Levenshtein Distance additionally takes the length of the text sequences into account and produces a similarity value between 0% and 100%, with 100% indicating that every single character needs to be changed and 0% indicating that both text sequences are identical. Lastly, we include the similarity analysis tool NiCad [50] into our comparison process to evaluate to which degree the prior application of bytecode normalization can improve the performance of similarity analysis tools. We use NiCad for our experiment, as it is one of the most popular similarity analysis tools.

Table 2 shows the JDKs considered in our evaluation. We considered all Java Long-Term-Support versions up to August 2023. According to a 2022 survey on the state of the Java ecosystem [44], our JDK selection covers more than 97% of JDK versions used in projects. This gives us an indication for the representativeness of our version selection. Furthermore, we considered the three most popular JDK vendors according to the survey, which include Oracle’s JDK, Amazon Corretto’s (AC) OpenJDK and Eclipse Adoptium’s (EA) OpenJDK, in our evaluation. Note that AC and EA do not distribute OpenJDK versions prior to version 8. Moreover, only a single project within our dataset can be compiled using Oracle’s JDK5 and JDK6, thus we do not consider these two no longer supported JDKs in our evaluation [26].

We executed the compilations and normalizations on a Debian 10 system, configured to use four cores of an Intel Xeon E5-2695 v3 (2.30 GHz) CPU and 32GB of main memory. We used Maven 3.8.6 for the invocation of builds.

4.2 RQ1: Does the JDK vendor influence the bytecode generation?

Before assessing the differences introduced when using different JDK or Java versions, we evaluate if different JDK *vendors* induce differences in the bytecode. Even though most vendors build upon the same OpenJDK source code, there are still some adjustments in regards to e.g. security fixes or performance improvements [1]. This research question aims at determining whether these changes may affect the generated bytecode. To do so, we compiled our full dataset of Java projects using the compilers of the JDK's listed in Table 2. Subsequently we compared all bytecode files generated by the compilers of the investigated vendors, using the same Java and JDK version, against each other.

None of the generated bytecode files contain any difference related to the vendor of the JDK used for compilation. These results indicate that changing the JDK vendor does not influence the bytecode generation of the JDK's compiler.

Based on this result, we consider a single JDK vendor (Oracle) in the remaining research questions.

4.3 RQ2: How does jNorm perform on changing JDK versions?

To investigate jNORM's normalization performance on different JDK versions, we kept all compilation settings at the project's configured default values and only varied the used JDK version within our experimental setup (see Section 4.1). For this experiment we considered versions 7, 8, 11, and 17 of Oracle's JDK.

Table 3 shows the results of our comparisons. The first column shows the pair of JDK versions used to generate the different artifacts we consider (see Section 4.1), e.g., in the first row (sets of) artifacts generated with JDK7 have been compared to (sets of) artifacts obtained from JDK8. The number inside the parentheses indicates the amount of projects we were able to compile with the respective JDKs. As we were not able to compile every project with all JDKs in our experimental setup, the number of compared projects varies based on the successful builds for each JDK. Notice that only few projects could be compiled using JDK7. This is due to features introduced in Java 8 being very popular in modern projects, e.g., default interface functions, streams and lambda expressions. To isolate differences introduced by incremental version increases, we decided to compare a JDK version with the next higher version in our experimental setup. To confirm that we do not miss differences by only comparing incremental version increases, we initially performed a comparison of projects compiled with JDK7 and JDK17 and compared the resulting set to the union of all incremental comparisons. In total we were able to compile ten projects using each version of Oracle's JDK in our experimental setup, comprising 4,621 bytecode files. This analysis showed that the set of differences obtained when comparing JDK7 to JDK17 is equal to the union of the sets of differences obtained when comparing each incremental version increase, i.e., $D_{7 \rightarrow 8} \cup D_{8 \rightarrow 11} \cup D_{11 \rightarrow 17} = D_{7 \rightarrow 17}$ with $D_{i \rightarrow j}$ representing the set of files containing compilation differences when comparing bytecode files yielded by the JDK i and JDK j compilers. This comparison holds true for all five processed sets of artifacts (bytecode, plain, optimized, normalized and aggressively normalized Jimple), showing that a comparison of incremental version increases does not miss compilation differences.

Columns two and three show the accumulated results of the textual comparison on a file-level granularity. The remaining columns show the accumulated comparison results on a method-level granularity. Columns "Files" and "Methods" show the total amount of files and methods we managed to compile with the respective JDKs. The "Diffs" columns

Table 3 Normalization results for different JDK versions². Percentage in brackets indicates the share of files/methods with compilation differences.

	Files	Diffs	Methods	Diffs	NLD	Disj.
JDK7 – JDK8 (29)						
Bytecode	8,060	1,058 (13.13%)	48,052	113 (0.24%)	4.02%	4
Jimple	8,069	93 (1.15%)	48,068	113 (0.24%)	5.78%	4
Optimized	8,069	93 (1.15%)	48,068	113 (0.24%)	5.90%	4
Normalized	8,069	24 (0.30%)	45,654	32 (0.07%)	8.13%	0
Aggressive	8,069	24 (0.30%)	45,654	32 (0.07%)	7.46%	0
JDK8 – JDK11 (98)						
Bytecode	45,625	8,068 (17.68%)	417,906	10,253 (2.45%)	9.00%	2,834
Jimple	60,265	2,959 (4.90%)	461,594	5,594 (1.21%)	9.31%	2,832
Optimized	60,265	2,852 (4.89%)	461,594	5,459 (1.18%)	8.28%	2,832
Normalized	60,265	995 (1.65%)	408,250	1,309 (0.32%)	4.80%	8
Aggressive	60,265	392 (0.65%)	408,250	426 (0.10%)	3.16%	8
JDK11 – JDK17 (91)						
Bytecode	58,623	13,936 (23.77%)	469,536	3,146 (0.67%)	18.94%	2,510
Jimple	80,584	2,566 (3.18%)	535,184	3,033 (0.57%)	17.30%	2,501
Optimized	80,584	2,553 (3.17%)	535,184	3,016 (0.56%)	17.06%	2,501
Normalized	80,584	120 (0.15%)	487,280	141 (0.03%)	4.24%	0
Aggressive	80,584	82 (0.10%)	487,280	95 (0.02%)	3.51%	0

show the total amount of files or methods that contained differences within the textual head-to-head comparison and their respective shares. The “NLD” column shows the average NLD of methods that contain differences, indicating the degree of dissimilarity induced by the compilation into individual methods. Note that only methods that are considered as not equal by the textual comparison are considered for the calculation of the NLD. The “Disj.” (disjunct) column represents the amount of methods that are present within the file generated by one JDK, but not within the file generated by the other JDK, e.g. synthetically generated bridge-methods. Therefore a direct comparison of such methods is not possible. Note that the transformation of bytecode to Jimple in some cases splits classes into multiple files, thus the number of bytecode files may differ from the number of Jimple files. In cases when dynamically invoked features like e.g. lambda functions are used, they are split into a separate file. We considered the by SOOT additionally generated files in our comparison.

One thing that is immediately noticeable is the high amount of differences in bytecode files when considering the file-level granularity, whereas the share of differences is considerably lower at method-level granularity. A detailed investigation into the differing bytecode files revealed this to be due to the presence of nested class information, which in bytecode is contained inside the class, but outside of methods. Depending on the used JDK, different modifiers are used or the order of these definitions varies. One can also observe that by simply converting the bytecode to Jimple, the amount of differences at file-level granularity considerably decreases. Nested class information, in contrast to the bytecode representation, is stored implicitly in the Jimple representation, which causes the disappearance of many dissimilarities. At method-level granularity, the amount of differences between bytecode and Jimple stays fairly similar. This indicates that besides removing some information

² Individual project results are available on <https://doi.org/10.5281/zenodo.12625104>

at class level, the plain conversion to Jimple itself does not significantly contribute to the normalization of method-level bytecode. Additionally it can be seen that the optimization step does not significantly contribute to the normalization by itself either. On the contrary, when looking at the results for the normalized file set, the amount of dissimilarities and disjunct methods heavily decreases. The remaining dissimilarities decrease even further when applying an aggressive normalization. Depending on the considered JDKs, the amount of compilation differences at file-level granularity decreases by up to 99% from the textual comparison of plain bytecode to the aggressively normalized Jimple. At method-level granularity the dissimilarity amount decreases by up to 97%.

We investigated the remaining differences in more detail to determine whether we missed other compilation difference classes. We found that the remaining differences are mostly due to more complex cases of compilation difference classes N4 and N8, which target try-catch blocks. Sometimes when such try-catch blocks are nested in specific ways, jNORM fails to apply the corresponding transformation correctly. Other differences are due to incorrect optimizations applied by the SOOT framework or an incorrect renaming of local variables.

Since we considered a small set of Java projects to establish the compilation difference classes in the first place and the evaluation across a large dataset of real-world Java projects only revealed a small set of edge cases of the already known difference classes not yet handled by jNORM, we believe that our normalization addresses the most common difference classes appearing within projects compiled with the investigated JDK versions. We will address the transformations incorrectly applied by jNORM in future work.

jNORM can remove up to 99% of the file-level differences and up to 97% of the method-level differences, which are induced when compiling the same source code with different versions of the JDK compiler.

4.4 RQ3: How does jNorm perform on changing Java target levels?

The target level that has been used to compile a specific Java class is typically included in the compiled class in form of a *major version* identifier [33]. While in some cases this information can be used to compile the source code to the version specified within the bytecode files, this is not possible when one wants to directly compare two already compiled bytecode files. Thus it is also important to assess jNORM’s performance on differing target levels.

To evaluate jNORM’s normalization performance on different Java target levels, we fixed the used JDK version and adjusted the target level in each project’s build configuration, within our experimental setup (see Section 4.1). All other build settings have been kept at each project’s provided configuration. We consistently used Oracle’s JDK11 in our experiment, as it is the most used JDK version in 2022 [44], which also offers backwards compatibility down to target level 6. To adjust the project’s target level we scanned each project of our dataset for build files (pom.xml). Inside each of the detected build files, we adjusted the *target*, *release*, or *java.version* properties, which are used to declare the desired Java target level [9, 8], to compile the project to our desired target levels. To validate that all projects were compiled with the intended target level, we verified the target level indicator [33] within the resulting bytecode files and removed it subsequently to not interfere with the comparison.

Table 4 shows the results of our experiment. The table has the same structure as Table 3, besides the first column now representing Java target levels instead of JDK versions. As in the previous experiment, we compare one target level to the next higher target level within our experimental setup, to best isolate compilation differences. To confirm that we do not miss differences by only comparing incremental version increases, we perform a comparison

Table 4 Normalization results for different target levels of the JDK11 compiler³. Percentage in brackets indicates the share of files/methods with compilation differences.

	Files	Diffs	Methods	Diffs	NLD	Disj.
T6 – T7 (25)						
Bytecode	13,774	29 (0.21%)	76,102	55 (0.07%)	2.38%	0
Jimple	14,411	29 (0.20%)	77,833	55 (0.07%)	2.36%	0
Optimized	14,411	29 (0.20%)	77,833	55 (0.07%)	2.32%	0
Normalized	14,411	8 (0.06%)	73,102	20 (0.03%)	0.56%	0
Aggressive	14,411	8 (0.06%)	73,102	20 (0.03%)	0.95%	0
T7 – T8 (31)						
Bytecode	4,100	70 (1.71%)	33,109	89 (0.27%)	1.90%	2
Jimple	4,127	33 (0.80%)	33,190	37 (0.11%)	5.05%	2
Optimized	4,127	33 (0.80%)	33,190	37 (0.11%)	4.67%	2
Normalized	4,127	32 (0.78%)	31,338	37 (0.12%)	4.67%	0
Aggressive	4,127	4 (0.10%)	31,338	4 (0.01%)	4.10%	0
T8 – T11 (80)						
Bytecode	28,709	13,260 (46.19%)	293,804	25,301 (8.61%)	18.39%	3,677
Jimple	42,690	9,654 (22.61%)	335,700	25,294 (7.53%)	17.74%	3,677
Optimized	42,690	9,654 (22.61%)	335,700	25,294 (7.53%)	17.58%	3,677
Normalized	42,690	110 (0.26%)	297,541	140 (0.47%)	2.76%	0
Aggressive	42,690	88 (0.21%)	297,541	115 (0.39%)	2.40%	0

of the maximum possible target level distance. Again, the following equation holds for the twelve projects (1,245 bytecode files) that can be compiled to each target level within our experimental setup, when using JDK11 $D_{11.6 \rightarrow 11.7} \cup D_{11.7 \rightarrow 11.8} \cup D_{11.8 \rightarrow 11.11} = D_{11.6 \rightarrow 11.11}$ with $D_{11.i \rightarrow 11.j}$ representing the set of files containing differences when comparing bytecode files yielded by JDK11 set to target levels i and j .

The number within the parentheses inside the first column indicates the amount of projects we were able to compile using JDK11 configured with the respective Java target levels. The total amount of successful builds is lower than the one we obtained in our previous experiment. This is due to the change in the provided build configuration that this experiment requires, which often leads to projects not being able to compile anymore.

One thing that is immediately noticeable in Table 4 is the low amount of compilation differences throughout target levels 6, 7 and 8. Many of these are removed by the (aggressive) normalization. We investigated the remaining differing files and discovered that all remaining differences are due to wrong type inference and incorrect renaming of local variables performed by the SOOT framework. The picture changes when considering a target level change from Java 8 to Java 11, as visible in the third row of Table 4. Almost half of the compared files show differences in a textual head-to-head comparison. This value decreases to around 10% when considering method-level granularity. Furthermore, the NLD is very high, indicating that the compiled methods are significantly dissimilar. This large amount of differences is due to many highly used features being affected by the target level increase. From Java target level 8 to 11 the way that string concatenation, private method calls, and inner classes are handled has been changed. These are features that are frequently used within Java

³ Individual project results are available on <https://doi.org/10.5281/zenodo.12625104>

projects. The conversion to Jimple removes around half of the differences at file-level, but removes only few differences on method-level. The subsequent optimization does not remove any differences in the textual comparison. The normalization, instead, heavily decreases the dissimilarities. After aggressive normalization the amount of dissimilarities decreases by more than 99.3% from the textual comparison of plain bytecode to aggressively normalized Jimple. On a method-level granularity the dissimilarity amount even decreases by 99.6%.

Again we performed a manual inspection of the remaining differing files and methods to uncover possibly missed compilation difference classes. The inspection showed that the remaining differences can mostly be attributed to incorrect optimizations by the SOOT framework and incorrect transformations of dynamic string concatenation and nest-based access control applied by JNORM in specific scenarios (e.g. boolean variables being handled as integer values in the string concatenation). Again, we leave addressing of incorrectly applied transformations by JNORM for future work.

As we already stated in the previous research questions, the absence of previously not uncovered compilation difference classes leads us to believe that we uncovered the most frequent compilation difference classes for projects compiled to the investigated Java target levels. Furthermore, we additionally investigated compilation environments in which we compared changes to the JDK version *and* target level, however, again we did not uncover any further difference classes.

JNORM can remove up to 99.3% of the file-level differences and up to 99.6% of the method-level differences, which are induced when compiling the same source code with different configured Java target levels.

4.5 RQ4: To which degree can bytecode normalization support similarity analysis tools?

In this research question we investigate whether existing similarity analysis tools are already capable of handling compilation differences on their own, without the need of a previous normalization by JNORM. To do so, we used our dataset from the previous research questions, but instead of performing a textual head-to-head comparison, we used the code clone detector NiCad 6.2 [50] to determine its performance without and with normalization applied. NiCad is a flexible and extensible code clone detector that is frequently used in similarity analysis studies. While there are other Java similarity analysis tools available, e.g. CCFinder [35], SourcererCC [52] or JPLAG [47], we decided on NiCad due to it being the most popular similarity analysis tool and its simple extensibility. It supports detection on block- or function-level granularities and different languages (e.g. Java, Rust, C) and applies its own pretty-printing and code normalizations before the similarity analysis. NiCad offers a wide array of possible applications like code clone or plagiarism detection and therefore provides many configurations. We employed NiCad set to function-level granularity with three different configurations (Plagiarism-1, Plagiarism-2 and Default) in our evaluation setup (described in Section 4.1). NiCad does not support Java bytecode nor Jimple out-of-the-box. To perform our experiment, we extended NiCad with the capability to handle Jimple inputs by adjusting and extending the provided Java grammar and normalization/transformation rules. Due to Jimple's syntactic similarity to Java source code, the only adjustments we performed on NiCad consisted of adding Jimple-exclusive statements (identity, invoke, goto, monitor), adjusting try-catch definitions and adjusting if-statements to their Jimple structure.

We decided to perform a plagiarism detection experiment since this is a typical scenario where only bytecode originating from an unknown compilation environment is available, but the corresponding source code is not. To perform the experiment we selected a method pair

Table 5 NiCad performance on method pairs with compilation differences. Percentage indicates the share of method pairs correctly identified as clones.

(a) JDK version comparison.

	Plag-1	Plag-2	Default
J7 – J8 (104)			
Jimple	12.5%	75.0%	99.0%
Optimized	14.4%	66.3%	99.0%
Normalized	78.9%	85.6%	99.0%
Agg. Normalized	81.7%	87.5%	100%
J8 – J11 (4,967)			
Jimple	35.2%	57.1%	90.2%
Optimized	38.6%	59.3%	92.1%
Normalized	82.9%	94.0%	99.9%
Agg. Normalized	97.4%	98.6%	99.9%
J11 – J17 (2,858)			
Jimple	6.4%	9.8%	26.1%
Optimized	6.9%	10.0%	25.9%
Normalized	96.7%	99.3%	99.7%
Agg. Normalized	98.0%	99.4%	99.9%

(b) Target level comparison.

	Plag-1	Plag-2	Default
T6 – T7 (53)			
Jimple	88.7%	94.3%	100%
Optimized	88.7%	94.3%	100%
Normalized	92.5%	98.1%	100%
Agg. Normalized	92.5%	98.1%	100%
T7 – T8 (37)			
Jimple	10.8%	86.5%	100%
Optimized	13.5%	83.8%	100%
Normalized	16.2%	83.8%	100%
Agg. Normalized	91.9%	100%	100%
T8 – T11 (24,095)			
Jimple	24.7%	28.3%	70.8%
Optimized	25.7%	28.0%	70.4%
Normalized	99.7%	99.9%	100%
Agg. Normalized	99.8%	99.9%	100%

that originated from the same source code, but was compiled within different compilation environments, and gave it to NiCad running with different configurations and checked whether it reported the pair as matching or not. Since each method pair originates from the exact same source code, NiCad *should* report it as plagiarism. We considered all methods that were not equal in the textual comparison within our Jimple dataset (see RQ2 4.3 & RQ3 4.4). We executed NiCad on the same set of methods across all other representations (optimized, normalized and aggressively normalized Jimple).

Tables 5a and 5b report the results of our experiment with NiCad. The first column of each table shows the compilation environments that the compared files originated from (J stands for JDK version and T stands for target level) and indicate which normalization steps have been applied before being forwarded to NiCad. The number in parentheses indicates the number of *method pairs* compared by NiCad. Note that, since NiCad was not able to analyze some pairs, the number is slightly lower than the differing methods reported in RQ2 and RQ3. The remaining columns show the detection recall of NiCad using different configurations. The Plagiarism-1 and Plagiarism-2 configurations are explicitly targeting plagiarism detection. The allowed degree of dissimilarity after applying pretty-printing and its own normalizations is specified at 10%. The Plagiarism-2 configuration additionally applies the *blind* renaming scheme, which removes identifier names. The provided Default configuration of NiCad does not target a plagiarism detection scenario but an aggressive code clone detection that allows a dissimilarity of up to 30%. It also applies the additional blind-renaming. Note that we test NiCad in a best-case scenario: First, we provide it with Jimple code, which in contrast to bytecode already contains fewer compilation differences. Second, we evaluate NiCad’s performance on small configuration changes only (e.g. comparing JDK7 to JDK8) and do not combine multiple configuration changes, which would result in even more compilation differences. Finally, as most similarity analysis studies [52] we do not take potential false positives into account. Especially with its aggressive Default configuration, which only requires 70% similarity for a match, NiCad would incorrectly classify method pairs as plagiarism cases, which actually do not originate from the same source code.

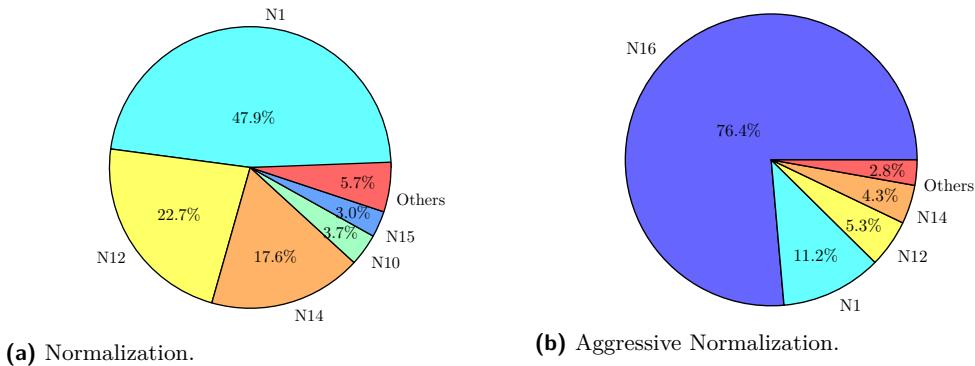


Figure 5 Average prevalence of the individual compilation difference classes.

One can observe that before applying normalization the Plagiarism-1 configuration performs poorly for most JDKs. However, after normalization is applied the performance drastically increases. A similar effect, although not as drastic, can be observed for Plagiarism-2. The aggressive Default configuration of NiCad performs well for compilation environments that do not contain a high degree of dissimilarity. However, for environment changes that actually induce significant differences in methods (J11 - J17 and T8 - T11), indicated by a high NLD between method pairs (see Tables 3 and 4), NiCad continues to perform poorly before normalization, but offers a significantly increased performance when normalizations are applied first via jNORM. Recall, that the Default configuration allows for up to 30% dissimilarity and is still not able to reliably classify method pairs as matching. Note that the provided code did not contain any intentional modifications, e.g. obfuscations, as the removal of such intentional modifications is not part of jNORM's scope. Even without intentional obfuscations, NiCad was not able to detect many of the clones.

NiCad, one of the most popular code clone detectors, is not capable of handling all differences induced by different compilation environments on its own. However, when applying normalization via jNORM first, NiCad's performance increases significantly.

4.6 RQ5: How prevalent are the individual compilation difference transformations of jNorm?

To investigate the prevalence of the compilation difference classes and their individual contribution to the normalization, we tracked each applied transformation within the normalization process of our dataset used throughout RQ2 – RQ4.

Figure 5 shows the average amount of transformations across each JDK and Java version setting. On average jNORM applies 888 transformations during normalization per project. One can see that a few of the established compilation difference classes make up the biggest share of the transformations. For plain normalization, transformation N1, which handles synthetically generated methods, makes up almost half of all transformations. This is due to the large amount of bridge-methods generated for each private method within nested classes. This transformation is followed by transformations N12, which normalizes string concatenations, and N14, which normalizes the invocation of private methods. Both of these are frequently used features within Java applications. The remaining transformations are only sparsely required. For aggressive normalization, transformation N16, which removes all typechecks, makes up by far the biggest share of all transformations. This is due to

the compiler frequently placing typechecks into the bytecode. This is further enhanced by our aggressive approach of removing *all* occurrences of typechecks. Since jNORM applies transformation N16 after all other transformations, the prevalence of the other compilation difference classes is the same as for plain normalization. During aggressive normalization 4,134 transformations are applied on average per project.

Transformations N16 (Insertion or removal of typechecks), N1 (Synthetically generated methods), N12 (Dynamic String concatenation) and N14 (Invocation of private methods) make up 97.2% of all applied transformations and are thus the most prevalent during normalization.

5 Related Work

Many similarity analysis approaches have been proposed, targeting source code, bytecode, or binary code, which typically come with their own set of normalizations.

Bytecode level. Only few approaches have been developed for bytecode similarity analysis. However, there are various scenarios in which Java source code is not available. Whenever this is the case, the comparison has to be performed on the bytecode. SeByte [36] is a similarity detector targeting Java bytecode. It divides the bytecode into tokens and separates them based on their types to employ the Jaccard similarity measure for matching. Baker and Manber [4] leverage a combination of the similarity comparison tools Diff, Siff and Dup to determine the degree of similarity of Java bytecode files. Yu et al. [58] use the Smith-Waterman algorithm to determine the similarity of two bytecode snippets. They extract instruction and method-call sequences from the bytecode and apply the Smith-Waterman algorithm to align the extracted sequences. Ji et al. [30] propose an approach to perform a plagiarism detection on bytecode. They divide the bytecode into sequences and utilize the adaptive local alignment to find potential plagiarisms. Davis and Godfrey [17] propose an approach to find clones that works on Assembler and bytecode. Their approach implements a greedy matching of instruction types and arguments by using an internal weight measure. Chen et al. [7] present an approach that aims at detecting application clones on Android markets. They utilize control flow graphs, to compare apps to each other and find clones in the Dalvik bytecode.

The above mentioned approaches do not explicitly mention how the differences in bytecode resulting from different compilation environments.

Source code level. For source code level similarity analysis many approaches have been developed. NiCad [50, 10] is a textual based code clone detector that targets a variety of programming languages. It uses different means of normalization and is designed to be easily extensible. CCFinder [35] transforms the input source code into a set of tokens and performs the comparisons on this set of tokens. SourcererCC [52] uses a similar token-based detection approach. However, SourcererCC specifically aims at high scalability and is optimized towards a usage on large software repositories. JPlag [47] divides the source code into token strings and applies a greedy string tiling algorithm to find plagiarisms within sets of applications. DECKARD [31] leverages the Abstract Syntax Tree representation of an application’s source code to perform the similarity analysis. StoneDetector [2] uses a more specialized code-representation called dominator trees, a concept often used in compilers, to detect structural clones, which use different syntactical constructs to implement the same

control flow. DeepSim [60] uses a deep learning model to find semantic similarities within code snippets that are syntactically different. Oreo [51] is another code clone detection tool that leverages deep learning. It uses a pre-trained model that utilizes several code metrics to decide whether two code snippets are clones of each other, even if their syntactical similarity is below 70%.

Source code based similarity analysis approaches have become much more permissive to syntactic differences over the years. This allows some tools to perform a similarity analysis across intermediate representations that are syntactically similar to the targeted source code. Selim et al. [53] investigated how the additional supplementation of the Jimple intermediate representation, alongside the source code, of a Java application can help in code clone detection tasks. To do so, they applied the clone detection tools CCFinder and Simian to Jimple code, which is syntactically similar to Java source code.

Ragkhitwetsagul et al. [49] evaluate and compare 30 different code similarity detection techniques, including code clone detectors, plagiarism detectors and compression tools, within different similarity analysis scenarios.

Binary level. As machine code is usually at the hardware level and there is a lot of variety in compilation environments and optimization levels, binary similarity analysis is a complex problem. David et al. [14, 15, 16] propose multiple approaches that decompose the assembly code of the binary into strands, which encode specific semantic behaviors in small units. Before comparing the units, in a similar way as JNORM and SootDiff do to achieve a more normalized representation, they transform the units to LLVM-IR and apply some optimizations and transformations to them, which are specific to LLVM-IR. Luo et al. [40] model the semantics of binaries with a set of symbolic formulas that represent input-output relations and use a theorem solver to determine their similarity. Hemel et al. [23] created the Binary Analysis Tool which uses different comparison strategies, like string matching, compression and a binary delta check to find software license violations within binaries. Many approaches like SAFE [42] and Xu et al.'s approach [57] use machine learning to determine the similarity of binaries. Marcelli et al. [41] investigate and compare multiple machine learning based approaches that try to classify the similarity of binaries. Haq and Caballero [21] present a survey of binary code similarity in which they analyze and systemically categorize 70 different binary similarity analysis approaches developed since 1999.

While most binary similarity analysis techniques cannot be directly applied towards bytecode similarity analysis, they can theoretically at least be adapted to it.

Compiler influence There are few works that investigate the relation of compilers to similarity analysis. Kononenko et al. [38] investigate a compilation's degree of influence on code clone detection. To do so, they compare the detected code clones within Java source code and bytecode compiled from the same source code which results in different sets of detected clones. Ragkhitwetsagul and Krinke [48] investigate how compilation and decompilation can influence the clone detection performance within Java code bases. They suggest that decompilation can aid as a complementary measure to source code based clone detection, but is not sufficient on its own. Dann et al. [12] investigate the impact that different compilation environments have on the resulting bytecode. They propose the bytecode comparison tool SootDiff, which employs an approach similar to JNORM, however only support one of the transformations we defined (string constant concatenation) and only considers Java versions 5 to 8. Xiong et al [56] investigate sources of non-determinism in the Java build process that hinder builds from being reproducible. They uncover 14 patterns that may introduce

non-equivalences in the build and present corresponding mitigation strategies. In the context of `JNORM` many of these sources of non-determinism are addressed by the conversion of bytecode to `Jimple`.

6 Threats to Validity

For our evaluation of `JNORM` we exclusively relied on projects that use Maven as build tool. While other Java build tools such as Gradle [20] exist, Maven is the most popular [29]. Furthermore, we limited our experiments to Java versions 5–8, 11 and 17. Although this version range covers all LTS-versions (up to August 2023) and are also the by far most frequently used Java versions in projects (see electronic appendix), other versions may yield different evaluation results and cause unidentified compilation differences. While we included a large number of Java projects in our evaluation, there may be some rare instances of differences induced by different compilation environments, which we did not uncover across our data set. Moreover, we tried to isolate the detected compilation difference classes to the specific configuration change they are caused by. There may also be other configuration changes that cause the same differences. However, as `JNORM` does not know the used configuration anyway, the differences will still be normalized.

Even though `JNORM`, in its default mode, only transforms constructs from the version produced within one compilation environment to the version produced in another environment, there still may be semantic changes created by the applied transformation. Furthermore, in few cases there are incorrect analyses, transformations and optimizations applied by the `SOOT` framework before the application of `JNORM`'s transformations.

7 Conclusion

In this paper we presented the concept of bytecode normalization for code similarity analysis. Bytecode normalization addresses the problem of comparing the bytecode of Java applications that are compiled in different compilation environments. This is especially necessary when the source code is not available, like in plagiarism, copyright or vulnerability detection and SBOM creation scenarios. By converting bytecode into the intermediate representation `Jimple`, applying common optimizations, transforming remaining compilation differences and applying naming standardization, we create a representation that is always the same no matter the JDK and Java (LTS) version used to compile the source code. To do so, we identified and presented 16 compilation difference classes, which are induced in the bytecode when using different JDK and Java versions.

Based on the concept of bytecode normalization we implemented `JNORM`. Our evaluation on a large set of popular real-world Java projects showed that compiling equal Java source code within different compilation environments leads up to 46% of all generated bytecode files containing differences for single incremental version increases. By using `JNORM` we can reduce the number of compilation differences by more than 99%, for most of the investigated compilation environments. Furthermore, our evaluation of the similarity analyzer `NiCad` showed that the tool was not able to handle all compilation differences on its own, yet an application of bytecode normalization via `JNORM` prior to the similarity analysis significantly improved the tool's performance, showcasing the effectiveness of bytecode normalization.

`JNORM` creates a code representation that is independent of the environment the Java source code has been compiled in and thus lowers the required complexity for subsequent similarity analysis. This could potentially pave the way for further research in the field of bytecode similarity analysis, bringing it closer to the wide range of tools and techniques currently available for analyzing source code similarity.

References

- 1 Amazon Corretto 8. Accessed 2023-03-31. URL: <https://docs.aws.amazon.com/corretto/latest/corretto-8-ug/what-is-corretto-8.html>.
- 2 Wolfram Amme, Thomas S. Heinze, and André Schäfer. You look so different: Finding structural clones and subclones in java source code. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2021, Luxembourg, September 27 - October 1, 2021*, pages 70–80. IEEE, 2021.
- 3 ASM: Java bytecode manipulation and analysis framework. Accessed 2022-10-24. URL: <https://asm.ow2.io/>.
- 4 Brenda S. Baker and Udi Manber. Deducing similarities in java sources from bytecodes. In *1998 USENIX Annual Technical Conference, New Orleans, Louisiana, USA, June 15-19, 1998*. USENIX Association, 1998.
- 5 Musard Balliu, Benoit Baudry, Sofia Bobadilla, Mathias Ekstedt, Martin Monperrus, Javier Ron, Aman Sharma, Gabriel Skoglund, César Soto-Valero, and Martin Wittlinger. Challenges of producing software bill of materials for java. *IEEE Security & Privacy*, pages 2–13, 2023.
- 6 Executive Order on Improving the Nation’s Cybersecurity. Accessed 2023-09-12. URL: <https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity>.
- 7 Kai Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *36th International Conference on Software Engineering, ICSE ’14, Hyderabad, India - May 31 - June 07, 2014*, pages 175–186. ACM, 2014.
- 8 Apache Maven Compiler Plugin - Setting the -release of the Java Compiler. Accessed 2023-04-03. URL: <https://maven.apache.org/plugins/maven-compiler-plugin/examples/set-compiler-release.html>.
- 9 Apache Maven Compiler Plugin - Setting the -source and -target of the Java Compiler. Accessed 2023-04-03. URL: <https://maven.apache.org/plugins/maven-compiler-plugin/examples/set-compiler-source-and-target.html>.
- 10 James R. Cordy and Chanchal K. Roy. The nicad clone detector. In *The 19th IEEE International Conference on Program Comprehension, ICPC 2011, Kingston, ON, Canada, June 22-24, 2011*, pages 219–220. IEEE Computer Society, 2011.
- 11 Cyber Resilience Act. Accessed 2023-09-12. URL: <https://digital-strategy.ec.europa.eu/en/library/cyber-resilience-act>.
- 12 Andreas Dann, Ben Hermann, and Eric Bodden. Sootdiff: bytecode comparison across different java compilers. In *Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2019, Phoenix, AZ, USA, June 22, 2019*, pages 14–19. ACM, 2019.
- 13 Andreas Dann, Henrik Plate, Ben Hermann, Serena Elisa Ponta, and Eric Bodden. Identifying challenges for oss vulnerability scanners-a study & test suite. *IEEE Transactions on Software Engineering*, 48(9):3613–3625, 2021.
- 14 Yaniv David, Nimrod Partush, and Eran Yahav. Statistical similarity of binaries. *Acm Sigplan Notices*, 51(6):266–280, 2016.
- 15 Yaniv David, Nimrod Partush, and Eran Yahav. Similarity of binaries through re-optimization. In *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation*, pages 79–94, 2017.
- 16 Yaniv David, Nimrod Partush, and Eran Yahav. Firmup: Precise static detection of common vulnerabilities in firmware. *ACM SIGPLAN Notices*, 53(2):392–404, 2018.
- 17 Ian J. Davis and Michael W. Godfrey. From whence it came: Detecting source code clones by analyzing assembler. In *17th Working Conference on Reverse Engineering, WCRE 2010, 13-16 October 2010, Beverly, MA, USA*, pages 242–246. IEEE Computer Society, 2010.
- 18 JDK-6246854 : Unnecessary checkcast in generated code. Accessed 2022-10-28. URL: https://bugs.java.com/bugdatabase/view_bug.do?bug_id=6246854.

- 19 GNU Compiler for Java (GCJ). Accessed 2022-10-17. URL: <https://gcc.gnu.org/wiki/GCJ>.
- 20 Gradle Build Tool. Accessed 2022-11-07. URL: <https://gradle.org/>.
- 21 Irfan Ul Haq and Juan Caballero. A survey of binary code similarity. *ACM Comput. Surv.*, 54(3):51:1–51:38, 2022.
- 22 Foyzul Hassan, Shaikh Mostafa, Edmund S. L. Lam, and Xiaoyin Wang. Automatic building of java projects in software repositories: A study on feasibility and challenges. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2017, Toronto, ON, Canada, November 9-10, 2017*, pages 38–47. IEEE Computer Society, 2017.
- 23 Armin Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. Finding software license violations through binary code clone detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 63–72, 2011.
- 24 The Java HotSpot Performance Engine Architecture. Accessed 2022-10-14. URL: <https://www.oracle.com/java/technologies/whitepaper.html>.
- 25 JEP 280: Indify String Concatenation. Accessed 2022-10-27. URL: <https://openjdk.org/jeps/280>.
- 26 Oracle Java SE 6 and JRockit End of Support. Accessed 2022-12-12. URL: https://support.oracle.com/knowledge/Middleware/2244851_1.html.
- 27 JDK Release Notes. Accessed 2023-03-30. URL: <https://www.oracle.com/java/technologies/javase/jdk-relnotes-index.html>.
- 28 Eclipse Java development tools (JDT). Accessed 2022-10-17. URL: <https://www.eclipse.org/jdt/core/>.
- 29 The State of Developer Ecosystem 2023. Accessed 2023-12-15. URL: <https://www.jetbrains.com/lp/devcosystem-2023/java/>.
- 30 Jeong-Hoon Ji, Gyun Woo, and Hwan-Gue Cho. A plagiarism detection technique for java program using bytecode analysis. In *2008 third international conference on convergence and hybrid information technology*, volume 1, pages 1092–1098. IEEE, 2008.
- 31 Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stéphane Glondu. DECKARD: scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 96–105. IEEE Computer Society, 2007.
- 32 IBM Jikes Compiler for the Java Language. Accessed 2022-10-17. URL: <https://sourceforge.net/projects/jikes/>.
- 33 The ClassFile Structure. Accessed 2023-12-12. URL: <https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.1>.
- 34 Oracle JVM Specification - Chapter 4. The class File Format. Accessed 2023-04-03. URL: <https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.8>.
- 35 Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilingual token-based code clone detection system for large scale source code. *IEEE Trans. Software Eng.*, 28(7):654–670, 2002.
- 36 Iman Keivanloo, Chanchal Kumar Roy, and Juergen Rilling. Sebyte: Scalable clone and similarity search for bytecode. *Sci. Comput. Program.*, 95:426–444, 2014.
- 37 Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. VUDDY: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 595–614. IEEE Computer Society, 2017.
- 38 Oleksii Kononenko, Cheng Zhang, and Michael W. Godfrey. Compiling clones: What happens? In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 481–485. IEEE Computer Society, 2014.
- 39 Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering, WCRE'01, Stuttgart, Germany, October 2-5, 2001*, pages 301–309. IEEE Computer Society, 2001.

- 40 Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 389–400, 2014.
- 41 Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. How machine learning is solving the binary function similarity problem. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2099–2116, 2022.
- 42 Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. Safe: Self-attentive function embeddings for binary similarity. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 309–329. Springer, 2019.
- 43 JEP 181: Nest-Based Access Control. Accessed 2022-10-28. URL: <https://openjdk.org/jeps/181>.
- 44 2022 State of the Java Ecosystem Report. Accessed 2022-10-24. URL: <https://newrelic.com/resources/report/2022-state-of-java-ecosystem>.
- 45 The Java programming language Compiler Group. Accessed 2022-10-17. URL: <https://openjdk.org/groups/compiler/>.
- 46 The Java Language Environment - Chapter 4: Architecture Neutral, Portable, and Robust. Accessed 2022-10-17. URL: <https://www.oracle.com/java/technologies/architecture-neutral-portable-robust.html>.
- 47 Lutz Prechelt, Guido Malpohl, Michael Philippsen, et al. Finding plagiarisms among a set of programs with jplag. *J. Univers. Comput. Sci.*, 8(11):1016, 2002.
- 48 Chaiyong Ragkhitwetsagul and Jens Krinke. Using compilation/decompilation to enhance clone detection. In *2017 IEEE 11th International Workshop on Software Clones (IWSC)*, pages 1–7. IEEE, 2017.
- 49 Chaiyong Ragkhitwetsagul, Jens Krinke, and David Clark. A comparison of code similarity analysers. *Empir. Softw. Eng.*, 23(4):2464–2519, 2018.
- 50 Chanchal Kumar Roy and James R. Cordy. NICAD: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *The 16th IEEE International Conference on Program Comprehension, ICPC 2008, Amsterdam, The Netherlands, June 10-13, 2008*, pages 172–181. IEEE Computer Society, 2008.
- 51 Vaibhav Saini, Farima Farmahinifarhani, Yadong Lu, Pierre Baldi, and Cristina V. Lopes. Oreo: detection of clones in the twilight zone. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 354–365. ACM, 2018.
- 52 Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. Sourcererc: scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 1157–1168. ACM, 2016.
- 53 Gehan M. K. Selim, King Chun Foo, and Ying Zou. Enhancing source-based clone detection using intermediate representation. In *17th Working Conference on Reverse Engineering, WCRE 2010, 13-16 October 2010, Beverly, MA, USA*, pages 227–236. IEEE Computer Society, 2010.
- 54 Soot Options and Phases. Accessed 2022-10-17. URL: https://soot-oss.github.io/soot/docs/4.3.0/options/soot_options.html.
- 55 Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. Soot – A java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999, Mississauga, Ontario, Canada*, page 13. IBM, 1999.

- 56 Jiawen Xiong, Yong Shi, Boyuan Chen, Filipe R Cogo, and Zhen Ming Jiang. Towards build verifiability for java-based systems. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, pages 297–306, 2022.
- 57 Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376, 2017.
- 58 Dongjin Yu, Jiazha Yang, Xin Chen, and Jie Chen. Detecting java code clones based on bytecode sequence alignment. *IEEE Access*, 7:22421–22433, 2019.
- 59 Li Yujian and Liu Bo. A normalized levenshtein distance metric. *IEEE transactions on pattern analysis and machine intelligence*, 29(6):1091–1095, 2007.
- 60 Gang Zhao and Jeff Huang. Deepsim: deep learning code functional similarity. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 141–151. ACM, 2018.

Optimizing Layout of Recursive Datatypes with Marmoset

Or, Algorithms + Data Layouts = Efficient Programs

Vidush Singhal 

Purdue University, West Lafayette, IN, USA

Joseph Zullo 

Purdue University, West Lafayette, IN, USA

Michael Vollmer 

University of Kent, UK

Ryan Newton 

Purdue University, West Lafayette, IN, USA

Chaitanya Koparkar 

Indiana University, Bloomington, IN, USA

Artem Pelenitsyn 

Purdue University, West Lafayette, IN, USA

Mike Rainey 

Carnegie Mellon University, Pittsburgh, PA, USA

Milind Kulkarni 

Purdue University, West Lafayette, IN, USA

Abstract

While programmers know that memory representation of data structures can have significant effects on performance, compiler support to *optimize* the layout of those structures is an under-explored field. Prior work has optimized the layout of individual, *non-recursive* structures without considering how collections of those objects in linked or *recursive* data structures are laid out.

This work introduces MAMOSET, a compiler that optimizes the layouts of algebraic datatypes, with a special focus on producing highly optimized, *packed* data layouts where recursive structures can be traversed with minimal pointer chasing. MAMOSET performs an analysis of how a recursive ADT is used across functions to choose a *global* layout that promotes simple, strided access for that ADT in memory. It does so by building and solving a constraint system to minimize an abstract cost model, yielding a predicted efficient layout for the ADT. MAMOSET then builds on top of GIBBON, a prior compiler for packed, mostly-serial representations, to synthesize optimized ADTs. We show experimentally that MAMOSET is able to choose optimal layouts across a series of microbenchmarks and case studies, outperforming both GIBBON's baseline approach, as well as MLTON, a Standard ML compiler that uses traditional pointer-heavy representations.

2012 ACM Subject Classification Software and its engineering → Compilers; Software and its engineering → Software performance; Information systems → Data layout

Keywords and phrases Tree traversals, Compilers, Data layout optimization, Dense data layout

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.38

Related Version Full Version: <https://arxiv.org/abs/2405.17590> [22]

Supplementary Material Software (ECOOP 2024 Artifact Evaluation approved artifact):
<https://doi.org/10.4230/DARTS.10.2.21>

Funding This work was supported in part by NSF CCF-1908504, CCF-1919197, CCF-2216978, CCF-2119352, CCF-1909862 and EPSRC EP/X021173/1.

1 Introduction

Recursive data structures are readily available in most programming languages. Linked lists, search trees, tries and others provide efficient and flexible solutions to a wide class of problems – both in low-level languages with direct memory access (C, C++, Rust, Zig) as well as high-level ones (Java, C#, Python). Additionally, in the purely functional (or *persistent* [20]) setting, recursive, tree-like data structures largely replace array-based ones.

 © Vidush Singhal, Chaitanya Koparkar, Joseph Zullo, Artem Pelenitsyn, Michael Vollmer, Mike Rainey, Ryan Newton, and Milind Kulkarni;
licensed under Creative Commons License CC-BY 4.0
38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 38; pp. 38:1–38:28



Implementation details of recursive data structures are not necessarily known to application programmers, who can only hope that the library authors and the compiler achieve good performance. Sadly, recursive data structures are a hard optimization target.

High-level languages represent recursive structures with pointers to small objects allocated sparsely on the heap. An algorithm traversing such a *boxed* representation spends much time in pointer chasing, which is a painful operation for modern hardware architectures. Optimizing compilers for these languages and architectures have many strengths but optimizing memory representation of user-defined data structures is not among them. One alternative is resorting to manual memory management to achieve maximum performance, but it has the obvious drawback of leaving convenience and safety behind.

A radically different approach is representing recursive datatypes as dense structures (basically, arrays) with the help of a library or compiler. The GIBBON compiler tries to improve the performance of recursive data structures by embracing dense representations by default [25]. This choice has practical benefits for programmers: they no longer need to take control of low-level data representation and allocation to serialize linked structures; and rather than employing error-prone index arithmetic to access data, they let GIBBON automatically translate idiomatic data structure accesses into operations on the dense representation.

Dense representations are not a panacea, though. They can suffer a complementary problem due to their inflexibility. A particular serialization decision for a data structure made by the compiler can misalign with the behavior of functions accessing that data. Consider a tree laid out in left-to-right pre-order with a program that accesses that tree right-to-left. Rather than scanning straightforwardly through the structure, the program would have to jump back and forth through the buffer to access the necessary data.

One way to counter the inflexibility of dense representations is to introduce some pointers. For instance, GIBBON inserts *shortcut pointers* to allow random access to recursive structures [24]. But this defeats the purpose of a dense representation: not only are accesses no longer nicely strided through memory, but the pointers and pointer chasing of boxed data are back. Indeed, when GIBBON is presented with a program whose access patterns do not match the chosen data layout, the generated code can be *significantly slower* than a program with favorable access patterns.

Are we stuck with pointer chasing when processing recursive data structures? We present MAMMOSET as a counter example. MAMMOSET is our program analysis and transformation approach that spots misalignments of algorithms and data layouts and fixes them where possible. Thus, our slogan is:

Algorithms + Data Layouts = Efficient Programs

MAMMOSET analyzes the data access patterns of a program and synthesizes a data layout that corresponds to that behavior. It then rewrites the datatype and code to produce more efficient code that operates on a dense data representation in a way that matches access patterns. This co-optimization of datatype and code results in improved locality and, in the context of GIBBON, avoidance of shortcut pointers as much as possible.

We implement MAMMOSET as an extension to GIBBON – a compiler based on dense representations of datatypes. That way, MAMMOSET can be either a transparent compiler optimization, or semi-automated tool for exploring different layouts during the programmer’s optimization work. Our approach has general applicability because of the minimal and common nature of the core language: the core language of MAMMOSET is a simple first-order, monomorphic, strict, purely functional language. Thanks to the succinct core language, we manage to isolate MAMMOSET from GIBBON-specific, complicated (backend) mechanics of converting a program to operate on dense rather than boxed data.

Overall, in this paper:

- We provide a static analysis capturing the temporal access patterns of a function towards a datatype it processes. As a result of the analysis, we generate a *field access graph* that summarizes these patterns.
- We define a cost model that, together with the field access graph, enables formulating the field-ordering optimization problem as an integer linear program. We apply a linear solver to the problem and obtain optimal positions of fields in the datatype definition relative to the cost model.
- We extend the GIBBON compiler to synthesize new datatypes based on the solution to the optimization problem, and transform the program to use these new, optimized types, adjusting the code where necessary.
- Using a series of benchmarks, we show that our implementation, MAMOSET, can provide speedups of 1.14 to 54 times over the best prior work on dense representations, GIBBON. MAMOSET outperforms MLTON on these same benchmarks by a factor of 1.6 to 38.

2 Dense Representation: The Good, The Bad, and The Pointers

This section gives a refresher on dense representations of algebraic datatypes (Section 2.1) and, using an example, illustrates the performance challenges of picking a layout for a datatype’s dense representation (Section 2.2).

2.1 Overview

Algebraic datatypes (ADTs) are a powerful language-based technology. ADTs can express many complex data structures while, nevertheless, providing a pleasantly high level of abstraction for application programmers. The high-level specification of ADTs leaves space to experiment with low-level implementation strategies. Hence, we use ADTs and a purely functional setting for our exploration of performance implications of data layout.

In a conventional implementation of algebraic datatypes, accessing a value of a given ADT requires dereferencing a pointer to a heap object, then reading the header word, to get to the payload. Accessing the desired data may require multiple further pointer dereferences, as objects may contain pointers to other objects, requiring the unraveling of multiple layers of nesting. The whole process is often described as *pointer chasing*, a fitting name, especially when the work per payload element is low.

In a dense representation of ADTs, as implemented in GIBBON, the data constructor stores one byte for the constructor’s *tag*, followed immediately by its fields, in the hope of avoiding pointer chasing. Wherever possible, the tag value occurs inline in a bytestream that hosts multiple values. As a result, values tend to reside compactly in the heap using contiguous blocks of memory. This representation avoids or reduces pointer chasing and admits efficient linear traversals favored by modern hardware via prefetching and caching.

2.2 Running Example

The efficiency of traversals on dense representations of data structures largely depends on how well access patterns and layout match each other. Consider a datatype (already monomorphized) describing a sequence of posts in a blog¹:

¹ Throughout the paper we use a subset of Haskell syntax, which corresponds to the input language of the GIBBON compiler.

38:4 Optimizing Layout of Recursive Datatypes with Marmoset

```

emphKeyword :: String → BlogList → BlogList
emphKeyword keyword blogs = case blogs of
    Nil → Nil
    Blog content hashTags blogs' →
        case search keyword hashTags of
            True → let content' = emphContent content keyword
                    blogs'' = emphKeyword keyword blogs'
                    in Blog content' hashTags blogs''
            False → let blogs'' = emphKeyword keyword blogs'
                      in Blog content hashTags blogs''

```

Figure 1 Blog traversal motivating example.

```

data BlogList = Nil | Blog Content HashTags BlogList

```

A non-empty blog value stores a content field (a string representing the body of the blog post), a list of hash tags summarizing keywords of the blog post, and a pointer to the rest of the list.² The datatype has one point of recursion and several variable-length fields in the definition. To extend on this, Section 5.3 contains an example of tree-shaped data (two points of recursion, in particular) with a fixed-length field. The most general case of multiple points of recursion and variable-length fields is also handled by MAMOSET.

The most favorable traversal for the `Blog` datatype is the same as the order in which the fields appear in the datatype definition. In this case, GIBBON can assign the dense and pointer-free layout as shown in Figure 2a. Solid blue arrows connecting adjacent fields represent unconditional sequential accesses – i.e., reading a range of bytes in a buffer, and then reading the next consecutive range. Such a traversal will reap the benefits of locality.

On the other hand, consider a traversal with less efficient access patterns (Figure 1). The algorithm scans blog entries for a given keyword in `hashTags`. If the hash tags of a particular blog entry contain the keyword, the algorithm puts an emphasis on every occurrence of the keyword in the content field. In terms of access patterns, if we found a match in the hash tags field, subsequent accesses to the fields happen in order, as depicted in Figure 2b. Otherwise, the traversal skips over the content field, as depicted in Figure 2c.

Here we use red lines to represent accesses that must *skip over* some data between the current position in the buffer and the target data. Data may be constructed recursively and will not necessarily have a statically-known size, so finding the end of a piece of unneeded data requires scanning through that data in order to reach the target data. This extra traversal (parsing data just to find the end of it) requires an arbitrarily large amount of work because the `content` field has a variable, dynamically-allocated size.

Extra traversals that perform useful no work, like skipping over the `content` field above, can degrade the asymptotic efficiency of programs. One way to avoid such traversals is to use pointers. For instance, when GIBBON detects that it has to skip over intervening data, it changes the definition of the constructor by inserting *shortcut pointers*, which provide an exact memory address to skip to in constant time. For our example program, GIBBON introduces one shortcut pointer for the `HashTags` field, and another one for the tail of the list.

² In practice, you may want to reuse a standard list type, e.g.:

```

type BlogList = [Blog]
data Blog = Blog Content HashTags

```

but a typical compiler (including GIBBON) would specialize the parametric list type with the `Blog` type to arrive at an equivalent of the definition shown in the main text above.

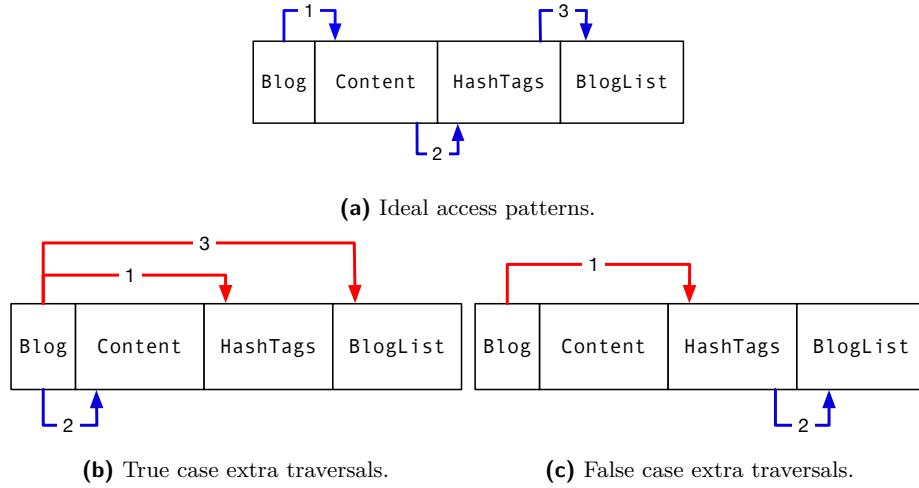


Figure 2 Showing a dense pointer-free layout with ideal accesses on top. Numbers represent the access order. Out of order accesses (red), incur costly extra traversals over fields in the middle.

The pointers provide direct accesses to the respective fields when needed and restore the constant-time asymptotic complexity for certain operations. This results in the access patterns shown in Figure 3a and 3c. Red dashed lines represent pointer-based constant-time field accesses. Otherwise, the access patterns are similar to what we had before.

The pointer-based approach in our example has two weaknesses. First, this approach is susceptible to the usual problems with pointer chasing. Second, just like with the initial solution, we access fields in an order that does not match the layout: the hash tags field is always accessed first but lives next to the content field.

MARMOSET, described in the following section, automates finding the weaknesses of the pointer-based approach and improving data layout and code accordingly. For instance, performance in our example can be improved by swapping the ordering between the `Content` and `HashTags` fields. Given this reordering, the hash tags are available directly at the start of the value, which lines up with the algorithm better, as the algorithm always accesses this field first. Additionally, our program’s `True` case (the keyword gets a hit within the hash tags) is more efficient because after traversing the content to highlight the keyword it stops at the next blog entry ready for the algorithm to make the recursive call. This improved data layout results in the more-streamlined access patterns shown in Figures 3b and 3d.

3 Design

MARMOSET infers efficient layouts for dense representations of recursive datatypes. MARMOSET’s key idea is that the best data layout should match the way a program accesses these data. Section 2 shows how this idea reduces to finding an ordering of fields in data constructors. The ordering must align with the order a function accesses those fields, in which case the optimization improves performance of the function.

To find a better layout for a datatype in the single-function case, MARMOSET first analyzes possible executions of the function and their potential for field accesses. In particular, MARMOSET takes into account (a) the various paths through a function, each of which may access fields in a different order, and (b) dependencies between operations in the function, as in the absence of dependencies, the function can be rewritten to access fields in the

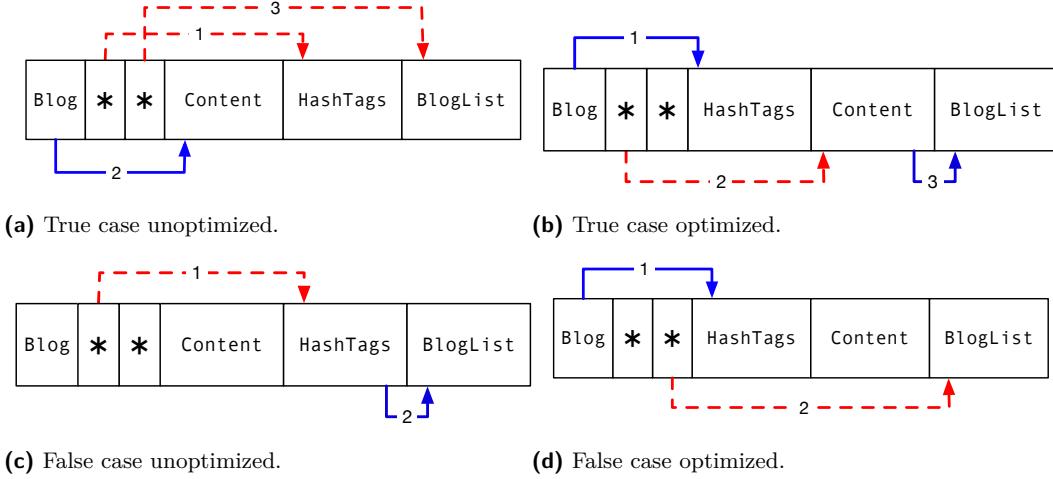


Figure 3 Showing the unoptimized representation with pointers to allow random access and the optimized layout with favorable access patterns.

original order, and that order will work best. MAMMOSSET thus constructs a control-flow graph (Section 3.2) and collects data-flow information (Section 3.3) to build a *field access graph*, a representation of the various possible orders in which a function might access fields (Section 3.4 and Section 3.5).

Once data accesses in a function are summarized in the field access graph, MAMMOSSET proceeds with synthesizing a data layout. MAMMOSSET incorporates knowledge about the benefits of sequential, strided access and the drawbacks of pointer chasing and backtracking to define an abstract cost model. The cost model allows to formulate an integer linear program whose optimal solution corresponds to a layout that minimizes the cost according to that model (Section 3.6).

The remainder of this section walks through this design in detail, and discusses how to extend the system to handle multiple functions that use a datatype (Section 3.7).

3.1 MAMMOSSET’s Language

MAMMOSSET operates on the language λ_M shown in our extended version [22]. λ_M is a first-order, monomorphic, call-by-value functional language with algebraic datatypes and pattern matching. Programs consist of a series of datatype definitions, function definitions, and a main expression. λ_M ’s expressions use A-normal form [12]. The notation \bar{x} denotes a vector $[x_1, \dots, x_n]$ and \bar{x}_i the item at position i . λ_M is an intermediate representation (IR) used towards the front end in the Gibbon compiler. The monomorphizer and specializer lower a program written in a polymorphic, higher-order subset of Haskell³ to λ_M , and then location inference is used to convert it to the *location calculus* (LoCal) code next [24]. It is easier to update the layout of types in λ_M compared to LoCal, as in λ_M the layout is implicitly determined by the ordering of fields, whereas the later LoCal IR makes the layout explicit using locations and regions (essentially, buffers and pointer arithmetic).

³ With strict evaluation semantics using `-XStrict`.

3.2 Control-Flow Analysis

Algorithm 1 Control-Flow Graph Psuedocode.

```

1: Input
2:   exp: An expression in subset of  $\lambda_M$ 
3:   weight: The likelihood of exp executing (i.e., exp's inbound edge)
4: Output
5:   A tuple of list of cfg nodes and the node id.
6: function CONTROLFLOWGRAPH(exp, weight)
7:   let nodeId = genFreshId()
8:   switch exp do
9:     case LetE (v, ty, rhs) bod
10:      let (nodes, succId) = CONTROLFLOWGRAPH(bod, weight)
11:      let newNode = (nodeId, (LetRHS (v, ty, rhs), weight), [succId])
12:      return (nodes ++ newNode, nodeId)
13:   end case
14:   case CaseE scrt cases
15:     let (nodes, successors) = CFGCASE(weight/length(cases), cases)
16:     let newNode = (nodeId, (scrt, weight), successors)
17:     return (nodes ++ [newNode], nodeId)
18:   end case
19:   case VarE v
20:     let newNode = (nodeId, (v, weight), [])
21:     return ([newNode], nodeId)
22:   end case
23: end switch
24: end function

```

We construct a control-flow graph with sub-expressions, and let-bound RHS's (right hand sides) of λ_M as the nodes. Algorithm 1 shows the psuedocode for generating the control-flow graph. Because the syntax is flattened into A-normal form, there is no need to traverse within the RHS of a let expression. Edges between the nodes represent paths between expressions. The edges consist of *weights* (Line 11) that represent the likelihood of a particular path being taken. An edge between two nodes indicates the order of the evaluation of the program. A node corresponding to a `let`-binding (Line 9) contains the bound expression and has one outgoing edge to a node corresponding to the body expression. A `case` expression (Line 14) splits the control flow n -ways, where n is the number of pattern matches. Outgoing edges of a node for a `case` expression have weights associated with them that correspond to the likelihood of taking a particular branch in the program. Control flow terminates on a leaf λ_M expression: a variable reference, a data constructor or a function application.

Figure 4a shows the control-flow graph for the running example (Figure 1). Each node corresponds to a sub-expression of the function `emphKeyword`. The first `case` expression splits the control flow into two branches, corresponding to whether the input list of blogs is empty or not. The branch corresponding to the empty input list is assigned a probability α , and the other branch is assigned a probability $1 - \alpha$. The next node corresponds to the pattern match `Blog content hashTags blogs'`. Another two-way branch follows, corresponding to whether `keyword` occurs in the `content` of this blog or not. We assign the probabilities σ and $1 - \sigma$ to these branches respectively. Note that as a result, the corresponding edges in the CFG have weights $(1 - \alpha) * \sigma$ and $(1 - \alpha) * (1 - \sigma)$, as the likelihood of *reaching* that condition is $(1 - \alpha)$. Each of these branches terminate by creating a new blog entry with its content potentially updated. In the current model, α and σ are 0.5: they are uniformly distributed.

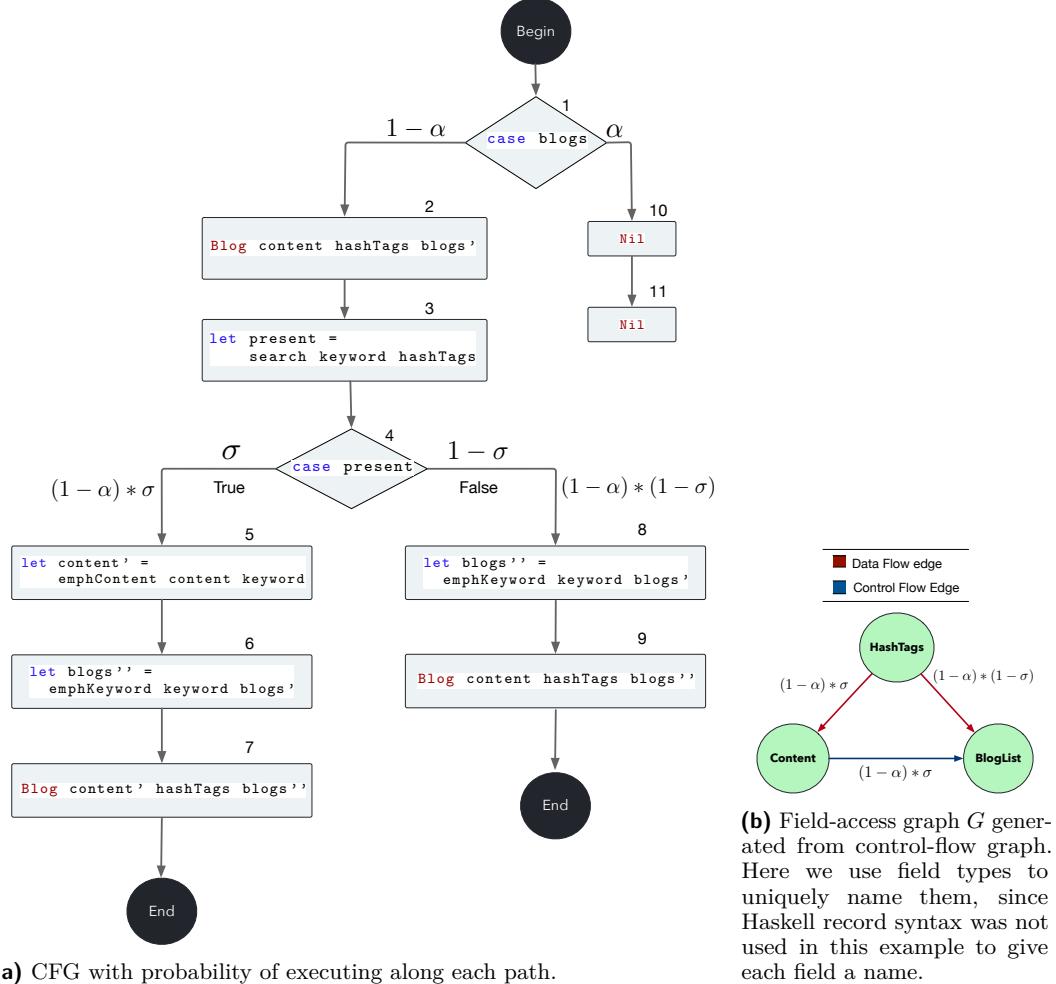


Figure 4 Control-flow and corresponding field access graphs generated for the running example.

One intuition for why realistic branch weights are not essential to MARMOSET’s optimization is that accurate weights only matter if there is a trade off between control-flow paths that are best served by different layouts. The base cases (e.g. empty list) typically contribute *no* ordering constraints, and in our experience, traversals tend to have a preferred order per function, rather than tradeoffs intra-function, which would reward having accurate, profile-driven branch probabilities. Hence, for now, we use uniform weights even when looking at the intra-function optimization.

3.3 Data Flow Analysis

We implement a straightforward analysis (use-def chain, and def-use chain) for `let` expressions to capture dependencies between `let` expressions. We use this dependence information to form dataflow edges in the field access graph (Section 3.5) and to subsequently optimize the layout and code of the traversal for performance. For *independent* `let` expressions in the function we are optimizing, we can transform the function body to have these let expressions in a different order. (Independent implies that there are no data dependencies between such `let` expressions. Changing the order of independent `let` expressions will not affect the

```

foo :: List → List
foo lst = case lst of
  Nil → Nil
  Cons x rst →
    let x' = x ^ 100
    rst' = foo rst
    in Cons x' rst'

```

(a) Function `foo` with `List`.

```

foo' :: List' → List'
foo' lst = case lst of
  Nil' → Nil'
  Cons' rst x →
    let rst' = foo' rst
    x' = x ^ 100
    in Cons' rst' x'

```

(b) Function `foo'` with `List'`.**Figure 5** Two different traversals on a list.

correctness of code, modulo exceptions.) However, we do such a transformation only when we deem it to be more cost efficient. In order to determine when re-ordering `let` expressions is more cost efficient, we classify fields based on specific attributes next.

3.4 Field Attributes For Code Motion

When trying to find the best layout, we may treat the code as immutable, but allowing ourselves to move the code around (i.e. change the order of accesses to the fields) unlocks more possibilities for optimizing layout. Not all code motions are valid due to data dependencies in the traversal. For instance, in a sequence of two `let` binders, the second one may reference the binding introduced in the first one: in this case, the two binders cannot be reordered.

To decide which code motions are allowed, we classify each field with one or more of the attributes: *recursive*, *scalar*, *self-recursive*, or *inlineable*. Some of these attributes are derived from the ADT definition and some from the code using the ADT. A *scalar* field refers to a datatype only made up of either other primitive types, such as `Int`. A *recursive* field refers to a datatype defined recursively. A *self-recursive* field is a recursive field that directly refers back to the datatype being defined (such as a `List` directly referencing itself). Finally, we call a field *inlineable* if the function being optimized makes a recursive call into this field (i.e. taking the field value as an argument). Hence, an *inlineable* field is necessarily a recursive field. As we show in the example below, the *inlineable* attribute is especially important when choosing whether or not to do code motion.

A single field can have multiple attributes. For instance, a function `traverse` doing a pre-order traversal on a `Tree` makes recursive calls on both the left and right children. The left and right children are recursive and self-recursive. Therefore, when looking at the scope of `traverse`, the left and right children have the attributes *recursive*, *self-recursive* and *inlineable*.

Example. Consider the example of a list traversal shown in Figure 5a. Here the function `foo` does some work on the `Int` field (it raises the `Int` to the power of 100) and then recurs on the tail of the list. Since MARMOSET compiles to dense representations, a `List`'s representation in memory stores the `Cons` tag (one byte) followed by the `Int` (8 bytes) followed by the next `Cons` tag and so on. Hence, the `Cons` tag and the `Int` field are interleaved together in memory. The function `foo` becomes a stream processor that consumes one stream in memory and produces a dense output buffer of the same type.

Alternatively, another layout of a list follows from the following definition:

```
data List' = Nil' | Cons' List' Int
```

In memory, the list has all `Cons'` tags next to each other (a unary encoding of array length!) and the `Int` elements all next to each other. In such a scenario, the performance of our traversal `foo` on the `List'` can improve traversal performance due to locality when accessing

38:10 Optimizing Layout of Recursive Datatypes with Marmoset

elements stored side-by-side⁴. However, this only works if we can subsequently change the function that traverses the list to do recursion on the tail of the list first and then call the exponentiation function on the `Int` field after the recursive call. If there are no data dependencies between the recursive call and the exponentiation function, then this is straightforward. We show `foo'` with the required code motion transformation to function `foo` accompanied with the change in the data representation from `List` to `List'` as shown in Figure 5b.

To optimize the layout, the tail of the list is assigned the attribute of `inlineable`. This attribute is used by the solver to determine a least-cost ordering to the `List` datatype in the scope of function `foo`. Whenever such code motion is possible, MAMOSET will place the `inlineable` field first and use code-motion to change the body of the function to perform recursion first if data flow dependencies allow such a transformation.

Structure of arrays. The transformation of `List/foo` to `List'/foo'` is similar to changing the representation of the `List` datatype to a *structure of arrays*, which causes the same types of values to be next to each other in memory. In particular, we switch from alternating constructor tags and integer values in memory to an array of constructor tags followed by an array of integers.

Note that the traversals `foo` and `foo'` have access patterns that are completely aligned with the data layout of `List` and `List'` respectively. The resulting speedup is solely a consequence of the *structure of arrays* effect. This is an added benefit to the runtime in addition to ensuring that the access patterns of a traversal are aligned with the data layout of the datatype it traverses.

3.5 Field Access Pattern Analysis

After constructing the CFG and DFG for a function definition, we utilize them to inspect the type of each of the function's input parameters – one data constructor at a time – and construct a *field-access graph* for it. Algorithm 2 shows the pseudocode for generating the field-access graph. This graph represents the temporal ordering of accesses among its fields.

The fields of the data constructor form the nodes of this graph. A directed edge from field f_i to field f_j is added if f_i is accessed immediately before f_j . Lines 13 to 24 in Algorithm 2 show how we keep track of the last accessed field and form an edge if possible. A directed edge can be of two different types. In addition, each edge has a associated weight which indicates the likelihood of accessing f_i before f_j , which is computed using the CFG. An edge can either be a data-flow edge or a control-flow edge (Lines 18 and 20). In Figure 4b, the red edge is a data flow edge and the blue edge is a control flow edge.

Data-Flow Edge indicates an access resulting from a data flow dependence between the fields f_i and f_j . In our source language, a data flow edge is induced by a `case` expression. A data flow edge implies that the code that represents the access is rigid in structure and changing it can make our transformation invalid.

Control-Flow Edge indicates an access that is not data-flow dependent. It is caused by the control flow of the program. Such an edge does not induce strict constraints on the code that induces the edge. The code is malleable in case of such accesses. This gives way to an

⁴ However, this effect can disappear if the elements are very large or the amount of work done per element becomes high, such that the percent of time loading the data is amortized.

Algorithm 2 Recursive function for generating the field access graph.

```

1: Input
2:   cur: current CFG node from which to start processing
3:   dcon: data constructor for which we are searching the best layout
4:   edges: field-access graph built so far
5:   lastAccessedVar: last accessed variable name, initially None
6:   dfgMap: set of data-flow edges between variables
7: Output
8:   Field access graph represented as a list of edges
9: function FIELDACCESSGRAPH(cur, dcon, edges, lastAccessedVar, dfgMap)
10:  let ((expr, weight), successors) = cur
11:  let mutable lastAccessedVarMut = lastAccessedVar
12:  let mutable edges' = edges
13:  for var : ORDEREDFREEVARIABLES(expr) do
14:    if !BOUNDPATTERNMATCHONDCON(var, dcon) then
15:      continue
16:    end if
17:    if lastAccessedVarMut != None then
18:      mutate edges' = ADDEdge(edges', ((lastAccessedVar, var), weight), ControlFlowTag)
19:      if LOOKUP((lastAccessedVar, var), dfgMap) then
20:        mutate edges' = ADDEdge(edges', ((lastAccessedVar, var), weight), DataFlowTag)
21:      end if
22:    end if
23:    mutate lastAccessedVarMut = var
24:  end for
25:  for succ : successors do
26:    let edges'' = FIELDACCESSGRAPH(succ, dcon, edges', lastAccessedVarMut, dfgMap)
27:    mutate edges' = MERGE(edges', edges'')
28:  end for
29:  return edges'
30: end function

```

optimization search space via code motion of `let` expressions. The optimization search space involves transformation of the source code, i.e, changing the access patterns at the source code level.

The field-access graph G is a directed graph, which consists of edges of the two types between fields of a datatype and can have cycles. The directed nature of the edges enforces a temporal relation between the corresponding fields. More concretely, assume that an edge e that connects two vertices representing fields fe_a (source of e) and fe_b (target of e). We interpret e as an evidence that field fe_a is accessed before field fe_b . The weight w for the edge e is the probability that this access will happen based on statically analyzing a function.

In our analysis, for a unique path through the traversal, we only account for the *first* access to any two fields. If two fields are accessed in a different order later on, the assumption is that the start address of the fields is likely to be in cache and hence it does not incur an expensive fetch call to memory. In fact, we tested our hypothesis by artificially making an example where say field f_a is accessed first, field f_b is accessed after f_a after which we constructed multiple artificial access edges from f_b to f_a , which might seem to suggest placing f_b before f_a . However, once the cache got warmed up and the start addresses of f_a and f_b are already in cache, the layout did not matter as much. This suggests that prioritizing for the first access edge between two unique fields along a unique path is sufficient for our analysis.

Two fields can be accessed in a different order along different paths through a traversal. This results in two edges between the fields. (The edges are in reversed order.) We allow at most two edges between any two vertices with the constraint that they have to be in the opposite direction and come from different paths in the traversal. If two fields are accessed in the same order along different paths in the traversal, we simply add the probabilities and merge the edges since they are in the same direction.

In order to construct G , we topologically sort the control-flow graph of a function and traverse it in the depth-first fashion via recursion on the successors of the current cfg node (Lines 25 to 28). As shown in line 14, we check if a variable is an alias to a field in the data constructor for which we are constructing the field-access graph G . As we process each node (i.e. a primitive expression such as a single function call), we update the graph for any direct or indirect references to input fields that we can detect. We ignore new variable bindings that refer to newly allocated rather than input data – they are not tracked in the access graph. We traverse the control-flow graph once, but we maintain the last-accessed information at each CFG node, so when we process a field access at an expression, we consult what was previously-accessed at the unique predecessor of the current CFG node. Figure 4b shows the generated access graph from the control-flow graph in Figure 4a. It also shows the probability along each edge obtained from the control-flow graph.

As we are traversing the nodes of the control-flow graph and generating directed edges in G , we use the likelihood of accessing that cfg node as the weight parameter (Line 10).

3.6 Finding a Layout

We use the field-access graph G to encode the problem of finding a better layout as an Integer Linear Program (ILP). Solving the problem yields a cost-optimal field order for the given pair of a data constructor and a function.

3.6.1 ILP Constraints

In our encoding, each field in the data constructor is represented by a variable, f_0, f_1, \dots . As a part of the result, each variable will be assigned a unique integer in the interval $[0, n - 1]$, where n is the number of fields. Intuitively, each variable represents an index in the sequence of fields.

The ILP uses several forms of constraints, including two forms of *hard* constraints:

$$\forall_{0 \leq i < n} \quad 0 \leq f_i < n \tag{1}$$

$$\forall_{0 \leq i < j < n} \quad f_i \neq f_j \tag{2}$$

The constraints of form 1 ensure that each field is mapped to a valid index, while the constraints of form 2 ensure that each field has a unique index. Constraints of either form must hold because each field must be in a valid location.

Hard constraints define valid field orderings but not all such reorderings improve efficiency, MAMOSET's main goal. To fulfil the goal, beside the hard constraints we introduce *soft* ones. Soft constraints come from the field access analysis. For example, assume that based on the access pattern of a function, we would *prefer* that field a goes before field b . We turn such a wish into a constraint. If the constraint cannot be satisfied, it will not break the correctness. In other words, such constraints can be broken, and that is why we call them soft.

3.6.2 Cost Model

MARMOSET encodes these soft constraints in the form of an abstract cost model that assigns a cost to a given layout (assignment of fields to positions) based on how efficient it is expected to be given the field-access graph.

To understand the intuition behind the cost model, note that the existence of an edge from field f_i to field f_j in the field-access graph means that there exists at least one path in the control-flow graph where f_i is accessed and f_j is the next field of the data constructor that is accessed. In other words, the existence of such an edge implies a preference *for that control-flow path* for field f_i to be immediately before field f_j in the layout so that the program can continue a linear scan through the packed buffer. Failing that, it would be preferable for f_j to be “ahead” of f_i in the layout so the program does not have to backtrack in the buffer. We can thus consider the costs of the different layout possibilities of f_i and f_j :

C_{succ} (f_j immediately after f_i): This is the best case scenario: the program traverses f_i and then uses f_j .

C_{after} (f_j after f_i in the buffer): If f_j is after f_i , but not *immediately* after, then the code can proceed without backtracking through the buffer, but the intervening data means that either a shortcut pointer or a extra traversal must be used to reach f_j , adding overhead.

C_{pred} (f_j immediately before f_i): Here, f_j is *earlier* than f_i in the buffer. Thus, the program will have already skipped past f_j , and some backtracking will be necessary to reach it. This incurs *two* sources of overhead: skipping past f_j in the first place, and then backtracking to reach it again.

C_{before} (f_j before f_i in the buffer): If, instead, f_j is farther back in the buffer than f_i , then the cost of skipping back and forth is greater: in addition to the costs of pointer dereferencing, because the fields are far apart in the buffer, it is less likely f_j will have remained in cache (due to poorer spatial locality).

We note a few things. First, the *exact* values of each of these costs are hard to predict. The exact penalty a program would pay for jumping ahead or backtracking depends on a variety of factors such as cache sizes, number of registers, cache line sizes, etc.

However, we use our best intuition to statically predict these costs based on the previously generated access graph. Note the existence of two types of edges in our access graph. An edge can either be a data-flow edge or a control-flow edge. For a data flow edge, the code is rigid. Hence the only axis we have available for transformation is the datatype itself. For a data flow edge, the costs are showed in Eq 3. Here, we must respect the access patterns in the original code which lead to the costs in Eq 3.

$$C_{succ} < C_{after} < C_{pred} < C_{before} \quad (3)$$

Note that a control-flow edge signifies that the direction of access for an edge is transformable. We could reverse the access in the code without breaking the correctness of the code. We need to make a more *fine-grained* choice. This choice involves looking at the attributes of the fields and making a judgement about the costs given we know the attributes of the fields. As shown in Sec 3.4 we would like to have the field with an *inlineable* attribute placed first. Hence, in our cost model, if f_i is inlineable, then we follow the same costs in Eq 3. However, if f_j is inlineable and f_i is not, we would like f_j to be placed before f_i . For such a layout to endure, the costs should change to Eq 4. For other permutations of the attributes, we use costs that prioritize placing the inlineable field/s first.

$$C_{pred} < C_{before} < C_{succ} < C_{after} \quad (4)$$

3.6.3 Assigning Costs to Edges

MARMOSET uses the field-access graph and the cost model to construct an objective function for the ILP problem. Each edge in the access graph represents one pair of field accesses with a preferred order. Thus, for each edge $e = (i, j)$, MARMOSET can use the indices of the fields f_i and f_j to assign a cost, c_e , to that pair of accesses following the rules below.

- If f_j is right after f_i , then assign cost C_{succ} , i.e.: $(f_j - f_i) = 1 \implies c_e = C_{succ}$.
- If f_j is farther ahead of f_i , then assign cost C_{after} , i.e.: $(f_j - f_i) > 1 \implies c_e = C_{after}$.
- If f_j is immediately before f_i , then assign cost C_{pred} , i.e.: $(f_j - f_i) = -1 \implies c_e = C_{pred}$.
- And if f_j is farther before f_i , then assign cost C_{before} , i.e.: $(f_j - f_i) < -1 \implies c_e = C_{before}$.

The cost of each edge, c_e must be multiplied by the *likelihood* of that edge being exercised, p_e , which is also captured by edge weights in the field-access graph. Combining these gives us a total estimated cost for any particular field layout:

$$C = \sum_{e \in E} c_e \cdot p_e \quad (5)$$

This is the cost that our ILP attempts to minimize, subject to the hard constraints 1 and 2.

3.6.4 Greedy layout ordering

Finding an optimal layout using an external solver hurts compile times. To solve this tradeoff, we propose a simple algorithm that traverses the field access graph in a *greedy* fashion. The algorithm starts from the root node of the graph, which corresponds to the field accessed first in the function, and greedily visits the child nodes based on the edge weights. We fix the edge order for a control-flow edge as the original order and do not look at field attributes. However, after the greedy algorithm picks a layout we match the let expressions to the layout order to make sure the code matches the layout order. The greedy algorithm is potentially sub-optimal when it comes to finding the best performing layout; however, the compile time is fast.

3.7 Finding a global layout

A data constructor can be used across multiple functions, therefore, we need to find a layout order that is optimal globally. To do so, we take constraints for each function and data constructor pair and combine them uniformly, that is, a uniform weight for each function. We then feed the combined constraints to the solver to get a globally optimal layout for that data constructor. The global optimization finds a globally optimal layout for all data constructors in the program. Once the new global layout is chosen for a data constructor, we re-write the entire program such that each data constructor uses the optimized order of fields. In the evaluation, we use the global optimization. However, we only show the data constructor that constitutes the major part of the program.

3.8 Finding a layout for functions with conflicting access patterns

Consider the datatype definition `D` with two fields `A`, `B`:

```
data D = D A B
```

If two functions, `f1` and `f2`, access the fields of `D` in the opposite orders, we get conflicting access patterns for `D`. For instance, assume `f1` accesses `A` first and then `B`, while `f2` accesses

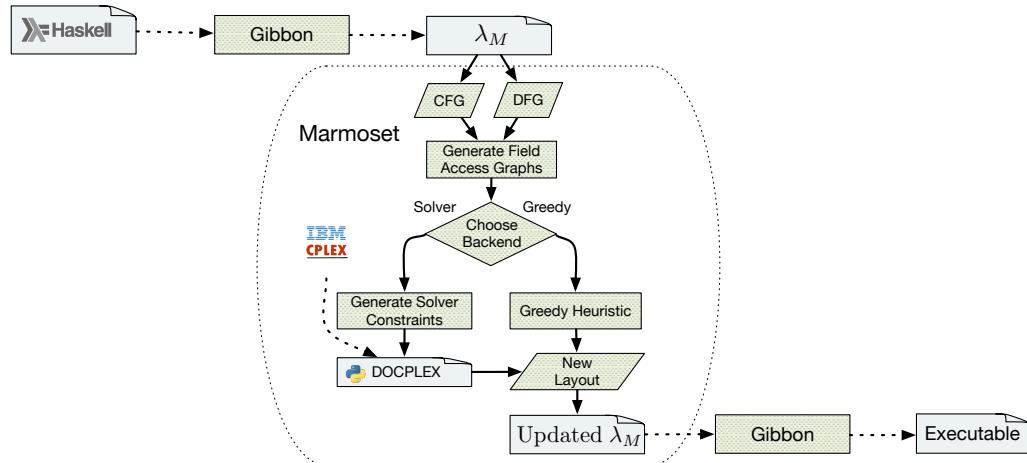


Figure 6 The overall pipeline of MAMOSET.

B first and then A. After combining edges across the two functions, we get two edges in opposition to each other. Since we use uniform weights for all functions, the edges will also have a uniform weight. As a result, placing A before B or vice versa are equally good in our cost model, and MAMOSET defers to the solver to get one of the two layouts.

With the two equally good layouts, MAMOSET’s solver (the default mode) chooses the layout favoring the function it picked first. For instance, if `f1` is defined earlier in the program, the order favoring `f1` will be picked. In the greedy mode, since both A and B are root nodes, MAMOSET will pick the first root node in the list of root nodes, which is, again, dependent on the ordering of functions in the source code.

4 Implementation

We implement MAMOSET in the open-source Gibbon compiler⁵. Figure 6 gives an overview of the overall pipeline. Gibbon is a whole-program micropass compiler that compiles a polymorphic, higher-order subset of (strict) Haskell.

The Gibbon front-end uses standard whole-program compilation and monomorphization techniques [7] to lower input programs into a first-order, monomorphic IR (λ_M). Gibbon performs location inference on this IR to convert it into a LoCal program, which has regions and locations, essentially, buffers and pointer arithmetic. Then a big middle section of the compiler is a series of LoCal \rightarrow LoCal compiler passes that perform various transformations. Finally, it generates C code. Our extension operates towards the front-end of the compiler, on λ_M . We closely follow the design described in Section 3 to construct the control-flow graph and field-access graph, and use the standard Haskell graph library⁶ in our implementation.

To solve the constraints, we use IBM’s DOCPLEX (Decision Optimization CPLEX), because its API allows high level modelling such as logical expressions like implications, negations, logical AND etc. with relatively low overhead. Unfortunately, there isn’t a readily available Haskell library that can interface with DOCPLEX. Thus, we use it via its library bindings available for Python. Specifically, we generate a Python program that feeds the constraints to DOCPLEX and outputs an optimum field ordering to the standard output, which MAMOSET reads and parses, and then reorders the fields accordingly.

⁵ <https://github.com/iu-parfunc/gibbon/>

⁶ <https://hackage.haskell.org/package/containers>

5 Evaluation

We evaluate MAMMOSET on three applications. First is a pair of microbenchmarks (Section 5.2) – a list length function and a logical expression evaluator – that help us explore performance penalties imposed by a sub-optimal data layout. Second is a small library of operations with binary trees (Section 5.3). Third is a blog management software based on the [BlogList](#) example from the Sections 2–3 (Section 5.4). Besides the run times, we take a closer look at how MAMMOSET affects cache behavior (Section 5.5) and compile times (Section 5.6). Finally, we discuss evaluation and its scale (Section 5.7).

We detail the impact of various datatype layouts on the performance. As the baseline, we use GIBBON, the most closely related prior work. We also compare MAMMOSET with MLTON (Section 5.4.2). For each benchmark, we run 99 iterations and report the run-time mean and the 95% confidence interval.

5.1 Experimental Setup

We run our benchmarks on a server-class machine with 64 CPUs, each with two threads. The CPU model is AMD Ryzen Threadripper 3990X with 2.2 GHz clock speed. The L1 cache size is 32 KB, L2 cache size is 512 KB and L3 cache size is 16 MB. We use GIBBON’s default C backend and call GCC 10.2.0 with `-O3` to generate binaries.

5.2 Micro Benchmarks

ListLength. This benchmark computes the length of a linked-list and demonstrates the cost of de-referencing memory addresses that are not present in the cache. It uses the linked list datatype:

```
data List = Nil | Cons Content List
```

If each element of the list is constructed using `Cons`, the traversal has to de-reference a pointer – to *jump over* the content – each time to access the tail of the list. This is an expensive operation, especially if the target memory address is not present in the cache. In contrast, if the `Content` and `List` fields were swapped, then to compute the length, the program only has to traverse n bytes for a list of length n – one byte per `Cons` tag – which is extremely efficient. Essentially, MAMMOSET transforms program to use the following datatype, while preserving its behavior:

```
data List' = Nil' | Cons' List' Content
```

In our experiment, the linked list is made of 3M elements and each element contains an instance of the *Pandoc Inline* datatype that occupies roughly 5KB. As seen in table 1⁷, the performance of the list constructed using the original `List` is $\sim 42 \times$ worse than the performance with the MAMMOSET-optimized, flipped layout. Not only does `List` have poor data locality and cache behavior, but it also has to execute more instructions to de-reference the pointer. Both M_{greedy} and M_{solver} choose the flipped layout `List'`.

LogicEval. This microbenchmark implements a short-circuiting logical expressions evaluator and runs it over synthetically generated, balanced syntax-trees with the height of 30. The intermediate nodes can be one of `Not`, `Or`, or `And`, selected at random, and the leaves hold boolean values. The syntax-tree datatype is defined as follows:

```
data Exp = Val Bool | Not Exp | Or Exp Exp | And Exp Exp
```

⁷ The performance of `List'` layout compiled with GIBBON differs from MAMMOSET as code motion to reorder let expressions results in different code. In addition to a noisy server.

Table 1 Run-time mean and 95% confidence interval (ub, lb) for different layouts (seconds). The last two columns show the run time for the layouts chosen by M_{greedy} and M_{solver} . The numbers in blue correspond to the lowest running time and the numbers in red correspond to the highest running time. Legend: l – left subtree, r – right subtree of the tree.

Benchmark name	GIBBON		MAMMOSET	
	List	List'	M_{greedy}	M_{solver}
ListLength	62.34 (62.26, 62.41)	1.51 (1.44, 1.59)	1.49 (1.41, 1.56)	1.50 (1.42, 1.58)
	lr	rl	M_{greedy}	M_{solver}
LogicEval	4.45 (4.42, 4.48)	6.60 (6.59, 6.61)	3.56 (3.53, 3.58)	3.55 (3.55, 3.55)
Rightmost	384.4 (368.4, 400.3)	314.5 (303.7, 325.3)	306.9 (295.6, 318.2)	303.1 (292.6, 313.6)

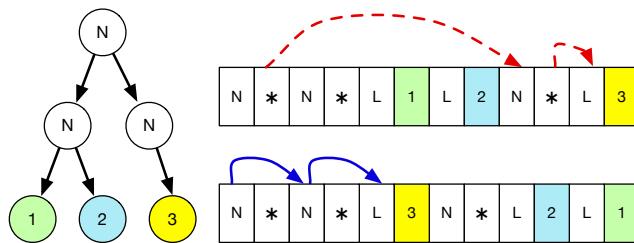


Figure 7 Rightmost: access patterns for, left-to-right (top) and right-to-left (bottom) serializations.

We measure the performance of the evaluator for different orders of the left and right subtrees. Since the short circuiting evaluates from left to right order of the `Exp`, changing the order of the left and right subtrees would affect the performance of the traversal. As can be seen in Table 1, the layout where the left subtree is serialized before the right subtree results in better performance compared to the tree where the right subtree is serialized before the left one. This is as expected since in the latter case, the traversal has to jump over the right subtree serialized before the left one in order to evaluate it first and then depending on the result of the left subtree possibly jump back to evaluate the right subtree. This results in poor spatial locality and hence worse performance. M_{greedy} and M_{solver} are able to identify the layout transformations that would give the best performance, which matches the case where the left subtree is serialized before the right subtree (Table 1⁸).

5.3 Binary Tree Benchmarks

We evaluate MAMMOSET on a few binary tree benchmarks: adding one to all values in a tree, exponentiation on integers stored in internal nodes, copying a tree and getting the right-most leaf value in the tree. For the first three benchmarks, the tree representation we use is:

```
data Tree = Leaf | Node Int Tree Tree
```

⁸ The layout chosen by M_{greedy} and M_{solver} is same as lr, the performance differs from the lr layout compiled with GIBBON as MAMMOSET does code motion which results in different code.

Table 2 Run-time mean and 95% confidence interval (ub, lb) for different layouts and traversal orders in the binary tree benchmarks (seconds). MISALGN_{pre} – post-order traversal on the pre-order layout of the tree. MISALGN_{post} – pre-order traversal on the post-order layout of the tree. ALGN_{pre} – pre-order traversal on the pre-order layout of the tree. ALGN_{in} – in-order traversal on an in-order layout of the tree. ALGN_{post} – post-order traversal on the post-order layout of the tree.

Benchmark name	GIBBON					MAMMOSET	
	MISALGN _{pre}	MISALGN _{post}	ALGN _{pre}	ALGN _{in}	ALGN _{post}	M _{greedy}	M _{solver}
AddOneTree	45.51 (45.35, 45.66)	memory error	1.29 (1.29, 1.29)	1.30 (1.30, 1.30)	1.29 (1.29, 1.29)	1.29 (1.29, 1.29)	1.28 (1.28, 1.28)
ExpTree	45.52 (45.34, 45.70)	memory error	1.31 (1.31, 1.31)	1.31 (1.31, 1.31)	1.29 (1.29, 1.29)	1.31 (1.31, 1.31)	1.29 (1.29, 1.29)
CopyTree	45.52 (45.37, 45.67)	memory error	1.29 (1.29, 1.29)	1.30 (1.30, 1.30)	1.28 (1.28, 1.28)	1.29 (1.29, 1.29)	1.28 (1.28, 1.28)

For right-most, the tree representation we use is:

```
data Tree = Leaf Int | Node Tree Tree
```

AddOneTree. This benchmark takes a full binary tree and increments the values stored in the internal nodes of the tree. We show the performance of an aligned preorder, inorder and postorder traversal in addition to a misaligned preorder and postorder traversal of the tree. Aligned traversals are ones where the data representation exactly matches the traversal order, for instance, a preorder traversal on a preorder representation of the tree. A misaligned traversal order is where the access patterns of the traversal don't match the data layout of the tree. For instance, a postorder traversal on a tree serialized in preorder. Table 2 shows the performance numbers. M_{solver} picks the aligned postorder traversal order which is best performing. It makes the recursive calls to the left and right children of the tree first and increments the values stored in the internal nodes once the recursive calls return. The tree representation is also changed to a postorder representation with the Int placed after the left and right children of the tree. This is in part due to the structure of arrays effect, as the Int are placed closer to each other. M_{greedy} on the other hand picks the aligned preorder traversal because of its greedy strategy which prioritizes placing the Int before the left and right subtree. The tree depth is set to 27. At this input size, the MISALGN_{post} traversal failed due to memory errors, and MISALGN_{pre} runs ~35× slower than aligned versions because of the skewed access patterns of the traversal.

ExpTree. This traversal does exponentiation on the values stored in the internal nodes of the tree. It is more computationally intensive than incrementing the value. We raise the Int to a power of 10 on a tree of depth 27. Table 2 shows the performance of the different layout and traversal orders. M_{solver} picks the ALGN_{post} representation which is the best performing, whereas M_{greedy} picks the ALGN_{pre} representation.

CopyTree. Copy-tree takes a full binary tree and makes a fresh copy of the tree in a new memory location. We use a tree of depth 27 in our evaluation. Table 2 shows the performance of different layout and traversal orders. We see that ALGN_{post} traversal performs the best. Indeed, M_{solver} picks the ALGN_{post} representation, whereas, M_{greedy} chooses the ALGN_{pre} representation.

Rightmost. This traversal does recursion on the right child of the tree and returns the Int value stored in the right-most leaf of the tree. Figure 7 shows an example of a tree with two different serializations of the tree: left-to-right (top) and right-to-left (bottom).

Table 3 Run-time mean and 95% confidence interval (ub, lb) for different layouts in the blog software benchmarks (seconds). Several possible permutations of layout are shown. Layout names abbreviations: h – Header, t – HashTags, b – Blogs, i – TagID, c – Content, a – Author, d – Date.

Bench.	GIBBON							MAMMOSET	
name	hiadctb	ctbhiad	tbchiad	tcbhiad	btchiad	bchiadt	cbiadht	M_{greedy}	M_{solver}
FilterBlogs									
	0.22 (0.22, 0.22)	0.22 (0.22, 0.22)	0.08 (0.08, 0.08)	0.27 (0.26, 0.27)	0.28 (0.28, 0.28)	0.29 (0.29, 0.30)	0.21 (0.21, 0.21)	0.07 (0.07, 0.07)	0.06 (0.06, 0.06)
EmphContent									
	0.67 (0.67, 0.67)	0.65 (0.65, 0.65)	1.60 (1.60, 1.60)	0.66 (0.66, 0.66)	1.63 (1.63, 1.63)	1.61 (1.61, 1.61)	0.47 (0.47, 0.47)	0.47 (0.47, 0.47)	0.64 (0.64, 0.64)
TagSearch									
	1.99 (1.99, 1.99)	1.98 (1.98, 1.98)	3.29 (3.29, 3.30)	1.68 (1.68, 1.68)	3.31 (3.31, 3.31)	3.30 (3.30, 3.30)	1.82 (1.82, 1.82)	1.76 (1.76, 1.76)	1.74 (1.74, 1.74)

The right-to-left serialization is more efficient because the constant-step movements (blue arrows) are usually more favorable than variable-step ones (red arrows) on modern hardware. Both M_{solver} and M_{greedy} pick the right-to-left serialization, and Table 1 shows that this choice performs better in the benchmark.

5.4 Blog Software Case Study

The Blog software case study serves as an example of a realistic benchmark, representing a sample of components from a blog management web service. The main data structure is a linked list of blogs where each blog contains fields such as header, ID, author information, content, hashtags and date. The fields are a mix of recursive and non-recursive datatypes. For instance, `Content` is a recursive type (the Pandoc `Block` type), but `Author` is a single string wrapped in a data constructor.⁹ One possible permutation of fields in the blog is:

```
data Blogs = Empty | HIADCTB Header Id Author Date Content HashTags Blogs
```

We evaluate MAMMOSET’s performance using three different traversals over a list of blogs. Overall, the traversals accept a keyword and a list of blogs; in the blogs, the traversals inspect either of the three fields: `Content`, `HashTags`, and the tail of the linked-list, `Blogs`. (Since `Blogs`, `Content`, and `HashTags` are recursive fields, changing their layout should represent greater differences in performance.) The fields used by an individual traversal are referred as *active fields* and the rest are referred as *passive fields*, and we specify these per-traversal below.

In Table 3, we report the performance of the six possible layouts obtained by permuting the order of the three recursive fields, and two additional layouts (Columns 1 and 2). The column names indicate the order of fields used; for example, the column `hiadctb` reports numbers for the layout with fields ordered as: `Header`, `Id`, `Author`, `Date`, `Content`, `HashTags`, and `Blogs`. All run times are gathered with GIBBON, and last two columns show the run times for code compiled using MAMMOSET’s greedy and solver-based optimization, respectively.

⁹ At times, we have to wrap scalars in data constructors to make them packed fields. GIBBON does not always support mixing scalar and packed fields due to compiler bugs.

FilterBlogs filters the list of blogs and only retains those which contain the given keyword in the `HashTags` field. The *active fields* for this traversal are `HashTags` and `Blogs`. Theoretically, the performance of this traversal is optimized when the `HashTags` field is serialized before `Blogs` on account of the first access to `HashTags` in the traversal. This is confirmed in practice with the layout ***tbchiad*** being the fastest. MAMOSET chooses the layout with `HashTags` serialized first and followed by `Blogs`; the order of other fields remains unchanged compared to the source program, but this has no effect on performance since they are *passive fields*. Table 3 also shows that layout chosen by MAMOSET performs similar to the layout ***tbchiad***. Both M_{solver} and M_{greedy} pick the layout ***tbhiadc*** when compiled from the initial layout ***hiadctb***.

EmphContent searches the content of each blog for the keyword and emphasizes all its occurrences there (if any). The active fields in this traversal are `Content` and `Blogs`. Based on the access pattern (`Content` accessed before `Blogs`), the layout with the best performance should place `Content` first followed by `Blogs`. In practice, the layout with the best performance is ***cbiadht***. In contrast, M_{solver} prioritizes the placement of `Blogs` before `Content`, but it also changes the traversal to recurse on the blogs first and then emphasize content. The passive fields are placed afterwards. The layout chosen by M_{solver} is ***bchiadt***, whereas the layout chosen by M_{greedy} is ***cbhiadt*** when compiled from the initial layout ***hiadctb***. The performance of M_{greedy} and M_{solver} differ because datatypes other than `Blogs` differ in their layout choices.

TagSearch looks for the presence of the keyword in the `HashTags` field, and if the keyword is present, the traversal emphasizes the keyword in the `Content`. The layout with the best performance is ***tcbhiad*** because of the access pattern, which inspects `HashTags` followed by `Content` followed by `Blogs`. M_{solver} chooses the layout ***tbchiad*** – which places `HashTags` followed by `Blogs` followed by `Content` – and changes the traversal to recurse on `Blogs` first and later emphasize `Content` in the `then` branch. On the other hand, M_{greedy} chooses ***tcbhiad*** when compiled from the initial layout ***hiadctb***.

5.4.1 Globally optimizing multiple functions

We use MAMOSET to globally optimize the three blog traversals we discussed above such that we pick one layout for all traversals that minimizes the overall runtime. Table 4 shows the runtime for a layout we compiled using GIBBON (***hiadctb***), M_{solver} (***tbchiad***) and M_{greedy} (***tbchiad***). We see that M_{greedy} and M_{solver} do a good job in reducing the traversal time globally. All the three traversals are run in a pipelined fashion sequentially. Although, M_{solver} does worse with `TagSearch` when run in a pipelined manner, it is actually better performing with M_{solver} when run alone as seen in table 3. Note that M_{solver} changes more than one data constructor based on the `inlineable` attribute that M_{greedy} does not. For instance, M_{solver} uses a packed `Inline` list in the `Content` with the tail serialized before `Inline`. Whereas, M_{greedy} uses a conventional packed list. In a pipelined execution of the traversals, although this helps reduce the runtime in the case of the content search traversal, it inadvertently increases the runtime in case of the tag search traversal due to a cache effect that can benefit from runtime information.

5.4.2 Comparison of MAMOSET against MLTON

We compare MAMOSET’s performance to MLTON, which compiles programs written in Standard ML, a strict language, to executables that are small with fast runtime performance.

Table 4 Run-time mean and 95% confidence interval (ub, lb) for the blog software benchmarks when MAMROSET optimizes the data layout globally (seconds). The input parameters are different from the single-function optimization case.

Benchmark name	GIBBON	MAMROSET	
	hiadctb	M _{greedy}	M _{solver}
FilterBlogs	2.23 (2.23, 2.23)	0.11 (0.11, 0.11)	0.10 (0.09, 0.10)
EmphContent	1.57 (1.57, 1.58)	1.38 (1.38, 1.38)	1.32 (1.32, 1.32)
TagSearch	2.20 (2.20, 2.20)	1.83 (1.83, 1.83)	2.35 (2.35, 2.35)

Table 5 PAPI performance counter statistics (average of 99 runs) for different blog traversals.

Benchmark name or metric	GIBBON						MAMROSET	
	hiadctb	ctbhiad	tbchiad	tcbhiad	btchiad	bchiadt	M _{greedy}	M _{solver}
FilterBlogs								
Ins	5.83e8	5.82e8	5.83e8	5.81e8	5.79e8	5.78e8	5.86e8	5.83e8
Cycles	9.89e8	9.60e8	2.85e8	1.05e9	1.20e9	1.25e9	8.78e8	2.79e8
L2 DCM	1.12e7	1.15e7	9.38e5	1.33e7	1.29e7	1.31e7	7.79e6	8.90e5
EmphContent								
Ins	5.85e9	5.83e9	6.78e9	5.84e9	6.78e9	6.78e9	5.84e9	5.84e9
Cycles	2.89e9	2.74e9	4.27e9	2.84e9	4.34e9	4.29e9	2.06e9	2.06e9
L2 DCM	1.28e7	1.08e7	2.03e7	1.33e7	2.05e7	2.10e7	7.73e6	7.66e6
TagSearch								
Ins	2.25e10	2.25e10	2.30e10	2.25e10	2.30e10	2.30e10	2.25e10	2.25e10
Cycles	8.59e9	8.59e9	9.61e9	7.29e9	9.61e9	9.75e9	7.88e9	7.61e9
L2 DCM	2.02e7	2.06e7	4.00e7	1.06e7	2.86e7	2.65e7	1.64e7	1.29e7

Figure 8 shows the speedup of MAMROSET over MLTON. As shown, the performance of MAMROSET is better than MLTON by significant margins for all the layouts and traversals. Since ADTs in MLTON are *boxed* – even though native integers or native arrays are *unboxed* – such a behavior is expected because it adds more instructions (pointer de-referencing) and results in worse spatial locality.

5.5 Cache behavior

The results from earlier sections demonstrate that MAMROSET’s layout choices improve run-time performance. This section investigates *why* performance improves. The basic premise of MAMROSET’s approach to layout optimization is to concentrate on minimizing how often a traversal needs to backtrack or skip ahead while processing a buffer. By minimizing this jumping around, we expect to see improvements from two possible sources. First, we expect to see an improvement in instruction counts, as an optimized layout should do less pointer

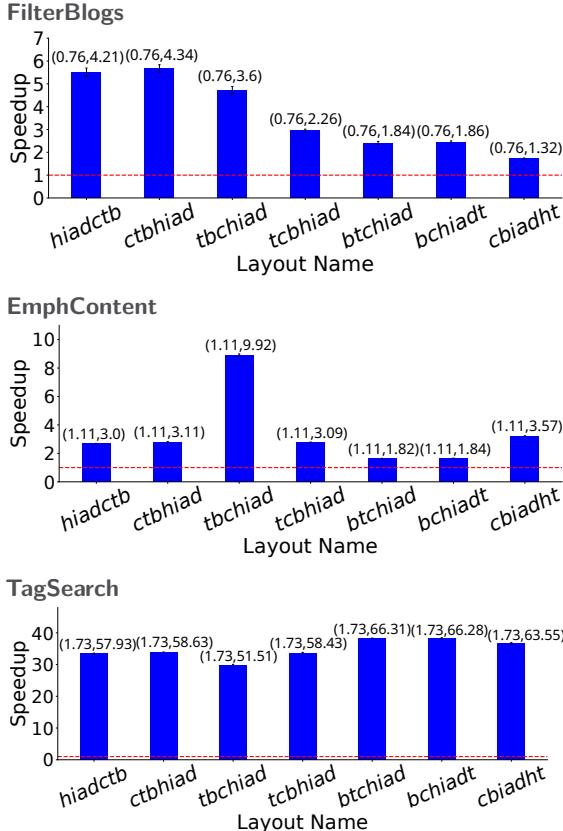


Figure 8 Performance comparison of M_{solver} with MLTON. The pair of numbers on top of each bar shows the median runtime in seconds of M_{solver} followed by the corresponding layout when compiled with MLTON.

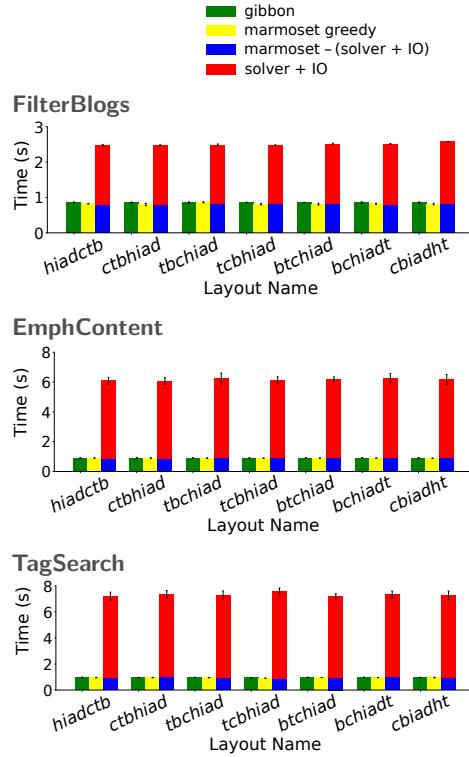


Figure 9 Average compile times (99 runs) in seconds for different layouts and traversal combinations when compiled with GIBBON and when optimized by M_{greedy} and M_{solver} .

chasing. Second, we expect to see an improvement in L2 and L3 cache utilization: both fewer misses (due to improved spatial locality and prefetching) and fewer accesses (due to improved locality in higher level caches).

Table 5 shows that our main hypothesis is borne out and the optimal layout has fewer L2 data cache misses¹⁰: a better layout promotes better locality. Interestingly, we do not observe a similar effect for instruction count. While different layouts differ in instruction counts, the difference is slight. We suspect this light effect of a better layout may be due to GIBBON’s current implementation, which often dereferences pointers even if a direct access in the buffer would suffice.

5.6 Tradeoffs between Marmoset’s solver and greedy optimization

To understand the difference between the layout chosen by M_{solver} and M_{greedy} we now take a closer look at the tag search traversal shown in Table 3. Here, both versions choose two different layouts with different performance implications. M_{solver} chooses the layout **tbchiad**

¹⁰ Since PAPI, the processor counters framework, does not completely support latest AMD processors yet, we were unable to obtain L3 cache misses, only the L2 DCM counter was available.

whereas M_{greedy} chooses the layout ***tcbhiad***. The mechanics of why can be explained using our running example (Figure 1) which is essentially a simplified version of the traversal shown in the evaluation. Figure 4b shows the access graph for this traversal. M_{solver} generates constraints outlined in Section 3.6 that lead to the layout ***tbchiad***. On the other hand, M_{greedy} starts at the root node of the graph and greedily chooses the next child to traverse. The order in which nodes of the graph are visited fixes the order of fields in the data constructor. In this case, the root node is `HashTags` which makes it the first field in the greedy layout, next, the greedy heuristic picks the `Content` field making it the second field and finally followed by the `BlogList` field.

In Figure 9 (p. 22), we show the compile times for different layout and traversal combinations when compiled with GIBBON, M_{greedy} and M_{solver} respectively as a measure of relative costs. The compile times for M_{solver} include the time to generate the control flow graph, the field access graph, the solver time and the time to re-order the datatype in the code. The solver times are in the order of the number of fields in a data constructor and not the program size. Hence, the solver adds relatively low overhead. Since the compiler does an IO call to the python solver, there is room for improvement in the future to lower these times. For instance, we could directly perform *FFI* calls to the CPLEX solver by lowering the constraints to C code. This would be faster and safer than the current implementation. In addition, during the global optimization, we call the solver on each data constructor as of the moment, we could further optimize this by sending constraints for all data constructors at once and doing just one solver call.

Although the cost of MAMOSET’s solver based optimization is higher than the greedy approach, it is a complementary approach which may help the user find a better layout at the cost of compile time. On the other hand, if the user wishes to optimize for the compile time, they should use the greedy heuristic.

5.7 Discussion: Scale of Evaluation

MAMOSET’s approach for finding the best layout for densely presented data is language agnostic, but the evaluation has to be language specific. Hence, we implemented the approach inside a most-developed (to our knowledge) compiler supporting dense representations of recursive datatypes, the Gibbon compiler. Our evaluation is heavily influenced by this.

At the time of writing, the scale of evaluation is limited by a number of Gibbon-related restrictions. Gibbon is meant as a tree traversal accelerator [25] and its original suite of benchmarks served as a basis and inspiration for evaluation of MAMOSET. “Big” end-to-end projects (e.g. compilers, web servers, etc.) have not been implemented in Gibbon and, therefore, are out of reach for us. If someone attempted to implement such a project using Gibbon, they would have to extend the compiler to support many realistic features: modules, FFI, general I/O, networking. Alternatively, one could integrate Gibbon into an existing realistic compiler as an optimization pass or a plugin. For instance, the Gibbon repository has some preliminary work for integrating as a GHC plugin¹¹, but it is far from completion. In any case, the corresponding effort is simply too big. Overall, the current MAMOSET evaluation shows that our approach is viable.

¹¹ <https://github.com/iu-parfunc/gibbon/tree/24c41c012a9c33bff160e54865e83a5d0d7867dd/gibbon-ghc-integration>

6 Future Work

MARMOSET could allow the user to provide optional constraints on the layout (either relative or absolute) through pragmas. A relative constraint would allow the user to specify if a field A comes immediately after field B. An absolute constraint would specify an exact index in the layout for a field. Such pragmas may be useful if the user requires a specific configuration of a data type for external reasons or has information about performance bottlenecks.

Although the performance optimization is currently statically driven, there are many avenues for future improvement. For instance, we can get better edges weights for the access graphs using dynamic profiling techniques. The profiling can be quite detailed, for instance, which branch in a function is more likely, which function takes the most time overall in a global setting (the optimization would bias the layout towards that function), how does a particular global layout affect the performance in case of a pipeline of functions.

We could also look at a scenario where we optimize each function locally and use “shim” functions that copy one layout to another (the one required by the next function in the pipeline). Although the cost of copying may be high, it warrants further investigation. Areas of improvement purely on the implementation side include optimizing whether MARMOSET dereferences a pointer to get to a field or uses the end-witness information as mentioned in section 2. Lesser pointer dereferencing can lower instruction counts and impact performance positively. We would also like to optimize the solver times as mentioned in 5.6.

We envision that the *structure of arrays* effect that we discovered may help with optimizations such as vectorization, where the performance can benefit significantly if the same datatype is close together in memory. Regardless, through the case studies, we see that MARMOSET shows promise in optimizing the layout of datatypes and may open up the optimization space for other complex optimizations such as vectorization.

7 Related Work

7.1 Cache-conscious data

Chilimbi and Larus [6] base on an object-oriented language with a generational garbage collector, which they extend with a heuristic for copying objects to the TO space. Their heuristic uses a special-purpose graph data structure, the *object affinity graph*, to identify when groups of objects are accessed by the program close together in time. When a given group of objects have high affinity in the object affinity graph, the collector is more likely to place them close together in the TO space. As such, a goal of their work and ours is to achieve higher data-access locality by carefully grouping together objects in the heap. However, a key difference from our work is that their approach bases its placement decisions on an object-affinity graph that is generated from profiling data, which is typically collected online by some compiler-inserted instrumentation. The placement decisions made by our approach are based on data collected by static analysis of the program. Such an approach has the advantage of not depending on the output of dynamic profiling, and therefore avoids the implementation challenges of dynamic profiling. A disadvantage of not using dynamic profiling is that the approach cannot adapt to changing access patterns that are highly input specific. We leave open for future work the possibility of getting the best of both approaches.

Chilimbi et al. [4] introduce the idea of hot/cold splitting of a data structure, where elements are categorized as being “hot” if accessed frequently and “cold” if accessed infrequently. This information is obtained by profiling the program. Cold fields are placed into a new object via an indirection and hot fields remain unchanged. In their approach, at runtime, there is a cache-conscious garbage collector [6] that co-locates the modified object instances.

This paper also suggests placing fields with high temporal affinity within the same cache block. For this they recommend `bbcach`, a field recommender for a data structure. `bbcach` forms a field affinity graph which combines static information about the source location of structure field accesses with dynamic information about the temporal ordering of accesses and their access frequency.

Chilimbi et al. [5] propose two techniques to solve the problem of poor reference locality. `ccmorp`. This works on tree-like data structures, and it relies on the programmer making a calculated guess about the safety of the operation on the tree-like data structure. It performs two major optimizations: clustering and coloring. Clustering takes a tree-like data structure and attempts to pack likely to be accessed elements in the structure within the same cache block. There are various ways to pack a subtree, including clustering k nodes in a subtree together, depth first clustering, etc. Coloring attempts to map simultaneously accessed data elements to non-conflicting regions of the cache.

`ccmalloc`. This is a memory allocator similar to `malloc` which takes an additional parameter that points to an existing data structure element which is likely accessed simultaneously. This requires programmer knowledge and effort in recognizing and then modifying the code with such a data element. `ccmalloc` tries to allocate the new data element as close to the existing data as possible, with the initial attempt being to allocate in the same cache block. It tries to put likely accessed elements on the same page in an attempt to improve TLB performance.

Franco et al. [13, 14] suggest that the layout of a data structure should be defined once at the point of initialization, and all further code that interacts with the structure should be “layout agnostic”. Ideally, this means that performance improvements involving layout changes can be made without requiring changes to program logic. To achieve this, classes are extended to support different layouts, and types carry layout information – code that operates on objects may be polymorphic over the layout details.

7.2 Data layout description and binary formats

Chen et al. [3] propose a data layout description framework *Dargent*, which allows programmers to specify the data layout of an ADT. It is built on top of the Cogent language [19], which is a first order polymorphic functional programming language. Dargent targets C code and provides proofs of formal correctness of the compiled C code with respect to the layout descriptions. Rather than having a compiler attempting to determine an efficient layout, their focus is on allowing the programmer to specify a particular layout they want and have confidence in the resulting C code.

Significant prior work went into generation of verified efficient code for interacting with binary data formats (parsing and validating). For example, EverParse [21] is a framework for generating verified parsers and formatters for binary data formats, and it has been used to formally verify zero-copy parsers for authenticated message formats. With Narcissus [9], encoders and decoders for binary formats could be verified and extracted, allowing researchers to certify the packet processing for a full internet protocol stack. Other work [23] has also explored the automatic generation of verified parsers and pretty printers given a specification of a binary data format, as well as the formal verification of a compiler for a subset of the Protocol Buffer serialization format [26].

Back [1] demonstrates how a domain-specific language for describing binary data formats could be useful for generating validators and for easier scripting and manipulation of the data from a high-level language like Java. [18] introduce *Packet Types* for programming with network protocol messages and provide language-level support for features commonly found in protocol formats like variable-sized and optional fields.

Hawkins et al. [15] introduce RELC, a framework for synthesizing low-level C++ code from a high-level relational representation of the code. The user describes and writes code that represents data at a high level as relations. Using a decomposition of the data that outlines memory representation, RELC synthesizes correct and efficient low-level code.

Baudon et al. [2] introduce the RIBBIT DSL, which allows programmers to describe the layout of ADTs that are monomorphic and immutable. RIBBIT provides a dual view on ADTs that allows both a high-level description of the ADT that the client code follows and a user-defined memory representation of the ADT for a fine-grained encoding of the layout. Precise control over memory layout allows RIBBIT authors to develop optimization algorithms over the ADTs, such as struct packing, bit stealing, pointer tagging, unboxing, etc. Although this approach enables improvements to layout of ADTs, it is different from MAMOSET's: RIBBIT focuses on *manually* defining low-level memory representation of the ADT whereas MAMOSET *automatically* optimizes the high-level layout (ordering of fields in the definition ADT) relying on GIBBON for efficient packing of the fields. While RIBBIT invites the programmer to encode their best guess about the optimal layout, MAMOSET comes up with such layout by analysing access patterns in the source code.

7.3 Memory layouts

Early work on specifications of memory layouts was explored in various studies of PADS, a language for describing ad hoc data-file formats [10, 11, 17].

Lattner and Adve [16] introduce a technique for improving the memory layout of the heap of a given C program. Their approach is to use the results of a custom static analysis to enable *pool allocation* of heap objects. Such automatic pool allocation bears some resemblance to our approach, where we use region-based allocation in tandem with region inference, and thanks to static analysis can group fields of a given struct into the same pool, thereby improving locality in certain circumstances.

Floorplan [8] is a declarative language for specifying high-level memory layouts, implemented as a compiler which generates Rust code. The language has forms for specifying sizes, alignments, and other features of chunks of memory in the heap, with the idea that any correct state of the heap can be derived from the Floorplan specification. It was successfully used to eliminate 55 out of 63 unsafe lines of code (all of the unsafe code relating to memory safety) of the immix garbage collector.

8 Conclusions

This paper introduces MAMOSET, which builds on GIBBON to generate efficient orders for algebraic datatypes. We show that a straightforward control-flow and data-flow analysis allows MAMOSET to identify opportunities to place fields of a data constructor near each other in memory to promote efficient consecutive access to those fields. Because a given function might use many fields in many different ways, MAMOSET adopts an approach of formulating the data layout problem as an ILP, with a cost model that assigns an abstract cost to a chosen layout. Armed with the ILP problem formulation, an off-the-shelf ILP solver allows MAMOSET to generate minimal-(abstract)-cost layout for algebraic datatypes. MAMOSET then uses the best layout to synthesize a new ADT and the GIBBON compiler toolchain to lower the code into an efficient program that operates over packed datatypes with minimal pointer chasing.

We show, across a number of benchmarks, that MAMOSET is able to effectively and consistently find the optimal data layout for a given combination of traversal function and ADT. In our experiments, MAMOSET-optimized layouts outperform not only GIBBON's default layouts but also the popular SML compiler MLTON.

References

- 1 Godmar Back. Datascript – A specification and scripting language for binary data. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, GPCE '02, pages 66–77, Berlin, Heidelberg, 2002. Springer-Verlag.
- 2 Thaïs Baudon, Gabriel Radanne, and Laure Gonnord. Bit-stealing made legal: Compilation for custom memory representations of algebraic data types. *Proc. ACM Program. Lang.*, 7(ICFP), August 2023. doi:[10.1145/3607858](https://doi.org/10.1145/3607858).
- 3 Zilin Chen, Ambroise Lafont, Liam O'Connor, Gabriele Keller, Craig McLaughlin, Vincent Jackson, and Christine Rizkallah. Dargent: A silver bullet for verified data layout refinement. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. doi:[10.1145/3571240](https://doi.org/10.1145/3571240).
- 4 Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 13–24, New York, NY, USA, 1999. Association for Computing Machinery. doi:[10.1145/301618.301635](https://doi.org/10.1145/301618.301635).
- 5 Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 1–12, New York, NY, USA, 1999. Association for Computing Machinery. doi:[10.1145/301618.301633](https://doi.org/10.1145/301618.301633).
- 6 Trishul M. Chilimbi and James R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *Proceedings of the 1st International Symposium on Memory Management*, ISMM '98, pages 37–48, New York, NY, USA, 1998. Association for Computing Machinery. doi:[10.1145/286860.286865](https://doi.org/10.1145/286860.286865).
- 7 Adam Chlipala. An optimizing compiler for a purely functional web-application language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 10–21, New York, NY, USA, 2015. ACM. doi:[10.1145/2784731.2784741](https://doi.org/10.1145/2784731.2784741).
- 8 Karl Cronburg and Samuel Z. Guyer. Floorplan: Spatial layout in memory management systems. In *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2019, pages 81–93, New York, NY, USA, 2019. Association for Computing Machinery. doi:[10.1145/3357765.3359519](https://doi.org/10.1145/3357765.3359519).
- 9 Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. Narcissus: Correct-by-construction derivation of decoders and encoders from binary formats. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019. doi:[10.1145/3341686](https://doi.org/10.1145/3341686).
- 10 Kathleen Fisher and Robert Gruber. Pads: A domain-specific language for processing ad hoc data. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 295–304, New York, NY, USA, 2005. Association for Computing Machinery. doi:[10.1145/1065010.1065046](https://doi.org/10.1145/1065010.1065046).
- 11 Kathleen Fisher and David Walker. The pads project: An overview. In *Proceedings of the 14th International Conference on Database Theory*, ICDT '11, pages 11–17, New York, NY, USA, 2011. Association for Computing Machinery. doi:[10.1145/1938551.1938556](https://doi.org/10.1145/1938551.1938556).
- 12 Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 237–247, New York, NY, USA, 1993. Association for Computing Machinery. doi:[10.1145/155090.155113](https://doi.org/10.1145/155090.155113).
- 13 Juliana Franco, Martin Hagelin, Tobias Wrigstad, Sophia Drossopoulou, and Susan Eisenbach. You can have it all: Abstraction and good cache performance. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2017, pages 148–167, New York, NY, USA, 2017. Association for Computing Machinery. doi:[10.1145/3133850.3133861](https://doi.org/10.1145/3133850.3133861).
- 14 Juliana Franco, Alexandros Tasos, Sophia Drossopoulou, Tobias Wrigstad, and Susan Eisenbach. Safely abstracting memory layouts, 2019. doi:[10.48550/arXiv.1901.08006](https://doi.org/10.48550/arXiv.1901.08006).
- 15 Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin Rinard, and Mooly Sagiv. Concurrent data representation synthesis. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 417–428, New York, NY, USA, 2012. Association for Computing Machinery. doi:[10.1145/2254064.2254114](https://doi.org/10.1145/2254064.2254114).

- 16 Chris Lattner and Vikram Adve. Automatic pool allocation for disjoint data structures. In *Proceedings of the 2002 Workshop on Memory System Performance*, MSP '02, pages 13–24, New York, NY, USA, 2002. Association for Computing Machinery. doi:10.1145/773146.773041.
- 17 Yitzhak Mandelbaum, Kathleen Fisher, David Walker, Mary Fernandez, and Artem Gleyzer. Pads/ml: A functional data description language. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 77–83, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1190216.1190231.
- 18 Peter J. McCann and Satish Chandra. Packet types: Abstract specification of network protocol messages. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '00, pages 321–333, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/347059.347563.
- 19 Liam O'Connor, Christine Rizkallah, Zilin Chen, Sidney Amani, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Alex Hixon, Gabriele Keller, Toby Murray, et al. Cogent: certified compilation for a functional systems language. *arXiv preprint arXiv:1601.05520*, 2016.
- 20 Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
- 21 Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. Everparse: Verified secure zero-copy parsers for authenticated message formats. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, pages 1465–1482, USA, 2019. USENIX Association.
- 22 Vidush Singhal, Chaitanya Koparkar, Joseph Zullo, Artem Pelenitsyn, Michael Vollmer, Mike Rainey, Ryan Newton, and Milind Kulkarni. Optimizing layout of recursive datatypes with marmoset, 2024. arXiv:2405.17590.
- 23 Marcell van Geest and Wouter Swierstra. Generic packet descriptions: Verified parsing and pretty printing of low-level data. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development*, TyDe 2017, pages 30–40, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3122975.3122979.
- 24 Michael Vollmer, Chaitanya Koparkar, Mike Rainey, Laith Sakka, Milind Kulkarni, and Ryan R. Newton. Local: A language for programs operating on serialized data. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 48–62, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3314221.3314631.
- 25 Michael Vollmer, Sarah Spall, Buddhika Chamith, Laith Sakka, Chaitanya Koparkar, Milind Kulkarni, Sam Tobin-Hochstadt, and Ryan R. Newton. Compiling Tree Transforms to Operate on Packed Representations. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, volume 74 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:29, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ECOOP.2017.26.
- 26 Qianchuan Ye and Benjamin Delaware. A verified protocol buffer compiler. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2019, pages 222–233, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293880.3294105.

Formalizing, Mechanizing, and Verifying Class-Based Refinement Types

Ke Sun 

Key Lab of HCST (PKU), MOE, School of Computer Science, Peking University, Beijing, China

Di Wang¹ 

Key Lab of HCST (PKU), MOE, School of Computer Science, Peking University, Beijing, China

Sheng Chen 

The Center for Advanced Computer Studies, University of Louisiana, Lafayette, LA, USA

Meng Wang 

University of Bristol, UK

Dan Hao 

Key Lab of HCST (PKU), MOE, School of Computer Science, Peking University, Beijing, China

Abstract

Refinement types have been extensively used in class-based languages to specify and verify fine-grained logical specifications. Despite the advances in practical aspects such as applicability and usability, two fundamental issues persist. First, the soundness of existing class-based refinement type systems is inadequately explored, casting doubts on their reliability. Second, the expressiveness of existing systems is limited, restricting the depiction of semantic properties related to object-oriented constructs. This work tackles these issues through a systematic framework. We formalize a declarative class-based refinement type calculus (named RFJ), that is expressive and concise. We rigorously develop the soundness meta-theory of this calculus, followed by its mechanization in Coq. Finally, to ensure the calculus's verifiability, we propose an algorithmic verification approach based on a fragment of first-order logic (named LFJ), and implement this approach as a type checker.

2012 ACM Subject Classification Theory of computation → Type structures; Software and its engineering → Formal software verification

Keywords and phrases Refinement Types, Program Verification, Object-oriented Programming

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.39

Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact)*:
<https://doi.org/10.4230/DARTS.10.2.22>

Funding This work is sponsored by National Natural Science Foundation of China Grant No. 62232001, NSF Grant 1750886, and EPSRC Grant EP/T008911/1.

Acknowledgements We thank the anonymous reviewers for their helpful comments.

1 Introduction

Refinement types have been widely used in class-based languages [47, 67, 10, 56, 26, 33, 36] to enhance the capabilities of traditional type systems, allowing for more precise safety guarantees. These types extend basic data types (e.g., integer type, boolean type, and class types) with logical constraints that specify detailed conditions on the data. For example, $\{\nu : C \mid \nu.f > 0\}$ characterizes instances of type C with the property that their f field exceeds zero. The logical constraint (e.g., $\nu.f > 0$) is often called the *refinement* of the type.

¹ Corresponding author

 © Ke Sun, Di Wang, Sheng Chen, Meng Wang, and Dan Hao;
licensed under Creative Commons License CC-BY 4.0
38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 39; pp. 39:1–39:30



Despite the advancements in various practical aspects (briefly surveyed in Section 8), a comprehensive examination of the fundamental aspects of class-based refinement types remains elusive. The primary reason stems from the intricate logical interpretation associated with refinements, which determines their meanings. In existing class-based refinement type systems, the logical interpretations are often defined via the Satisfiability Modulo Theories (SMT) relation [9, 35], since they are typically analyzed algorithmically via SMT solvers. Although this interpretation is closer to the actual algorithmic interpretation, it brings two crucial problems. Firstly, the **soundness** of the type system is difficult to define and argue formally, since it depends on the SMT relation, which is both complex and intricate to define properly. This complexity has led to a paucity of mechanized soundness proofs in previous systems, putting their reliability in doubt. Secondly, the **expressiveness** of the refinement types is limited by the need to adhere to decidable theory combinations (e.g., QF-EUFLIA [9]), which impedes the representation of semantic properties relevant to user-defined classes and methods.

To address those fundamental issues, this work makes three consecutive contributions.

1. Formalization. To formalize a foundational calculus, this paper introduces Refinement Featherweight Java (RFJ), an FJ-like [30] calculus with expressive refinements capable of stating arbitrary properties about user-defined elements. Our basic methodology is to construct a declarative, SMT-independent logical interpretation within the language, which greatly increases the expressiveness and benefits the meta-theoretical development. Apart from that, RFJ is equipped with several features important for refinement-type-based verification, yet mostly absent from previous systems, such as interfaces for decomposing proof obligations among subclasses (c.f., Section 2.2), general selfification – a typing mechanism [66, 51] (detailed in Section 2.1) that seamlessly integrates accurate term information into refinements, and flexible method overriding through co/contra-variance [11]. Besides those critical features, RFJ closely mirrors FJ, avoiding the usage of non-standard judgments (e.g., object constraint systems [47, 67]) and non-standard constructs (e.g., ANF [10], existential types [67]). Thus, we believe RFJ can be an ideal base to explore further extensions.

2. Mechanization. The soundness properties of RFJ are rigorously established and mechanized in Coq. Although leveraging an in-language logical interpretation reduces the proof difficulty, the proof is still challenging and requires non-standard techniques. For example, we make a novel use of big-step semantics to obtain a convenient induction principle for proving the preservation lemma under arbitrary type substitution. We introduce a novel approach to establish the logical soundness property (one major soundness property of RFJ), as the standard logical relation technique [62] is ineffective for first-order languages like RFJ [55].

3. Verification. The expressive refinements provided by RFJ fall out of the scope of existing SMT theories, casting ambiguity on the system’s algorithmic verifiability. We address this concern by proposing an algorithmic verification approach based on a fragment of order-sorted first-order logic (OS-FOL) [57]. We name this fragment as LFJ, and define a type-directed translation from RFJ to the LFJ. We define an intended model of LFJ and map the RFJ refinement subtyping problem to the LFJ validity problem under this model. We devise an axiomatization of the intended model covering the semantics of RFJ programs. The axiomatization can be used by SMT solvers to perform algorithmic verification. Thus, the expressive refinements of RFJ are not only meta-theoretical constructs: they are amenable to algorithmic analysis within SMT solvers. Additionally, we develop a refinement type

checker that leverages Z3 [19] for checking the validity of LFJ formulas. The type checker is evaluated against a small yet representative benchmark derived from a Java textbook [23] and prior systems [65].

In the remainder of this paper, we detail our contributions. Section 2 provides an overview. Sections 3, 4, and 5 each describe one of the three contributions. Section 6 discusses the mechanization and implementation. Sections 7, 8, and 9 review related work and conclude.

The accompanying code of this paper, including the meta-theory mechanization and type checker implementation, is available as the supplementary material of this paper.

2 Overview

This section serves as an overview of the whole paper. We start with an example program to demonstrate the expressiveness and features of RFJ. Then, we discuss the actual verification through LFJ. Finally, we turn back to the meta-theory of RFJ and the challenges of developing the meta-theory. The sequence of discussion – starting with verification before addressing meta-theoretical concerns – is intentionally chosen to contrast with the presentation order in subsequent sections, aiming to enhance comprehension by familiarizing readers with the system through its verification aspects first.

2.1 RFJ by Example

```

1  class Pizza{
2      {v:int|v>0} price(){return 1;}
3      Pizza remA(){return new Pizza();}
4      Pizza sell (this.price()>5){return this;}
5  class Crust extends Pizza{
6      {v:int|v>0} price(){return 1;}
7      Pizza remA(){return new Crust();}
8      Pizza sell (){return this;}
9  class Cheese extends Pizza{
10     p:Pizza
11     {v:int|v>0} price(){return let pp = this.p.price() in pp + 1;}
12     Pizza remA(){return new Cheese(this.p.remA());}
13  class Anchovy extends Pizza{
14     p:Pizza
15     {v:int|v>0 && v>=this.p.price()} price(){return let pp = this.p.price() in pp;}
16     Pizza remA(){return this.p.remA();}
17  class MagicAnchovy extends Anchovy{
18     {v:int|v>0 && v>this.p.price()} price(){return let pp = this.p.price() in pp + 1;}
19  class Main{
20     int assertSingleCheesePizza(x: {v:Pizza|v = new Cheese(new Crust())}){
21         return 0;
22     int testRemA(){
23         return let p1 = new Anchovy(new Cheese(new Crust())) in
24             this.assertSingleCheesePizza(p1.remA());}

```

Figure 1 An example RFJ program. In refinements, v stands for ν .

In this section, we illustrate RFJ using a program extended from a textbook example [23] (we add some methods to make it more interesting). The program models various pizzas and three operations on them: computing the price of a pizza (the `price` method), removing all anchovies from a pizza (the `remA` method), and selling a pizza (the `sell` method).

Simple Verification. Our initial focus is a basic property: the price of any pizza must be positive. To enforce this property, we refine the return types of `price` methods with a refinement $\nu > 0$, where ν denotes the value being refined. RFJ’s refinement subtyping mechanism guarantees that the `price` methods indeed return positive values. Pick `Pizza.price` for an example, RFJ enforces the following subtyping constraint for the return type:

$$\textit{this} : \{\nu : \textit{Pizza} | \textit{true}\} \vdash \{\nu : \textit{int} | \nu = 1\} <: \{\nu : \textit{int} | \nu > 0\} \quad (1)$$

In refinement type systems like RFJ, such subtyping constraints have logical interpretations. In particular, Constraint (1) requires all ν *satisfying* $\nu = 1$ must also *satisfy* $\nu > 0$, which holds under RFJ logical interpretation (formally defined in Section 3.3). Note that the subtyping constraint is checked within a specific type environment $\textit{this} : \{\nu : \textit{Pizza} | \textit{true}\}$, which contains the types of all visible variables. Those variables may be referred to by the refinement types, as demonstrated in the following example.

Method Override. RFJ supports overriding methods in subclasses. For example, `Cheese.price` overrides `Pizza.price` to provide a different price computation. To preserve the logical property, the return type must still be validated, yielding the following constraint:

$$\textit{this} : \{\nu : \textit{Cheese} | \textit{true}\}, \textit{pp} : \{\nu : \textit{int} | \nu > 0\} \vdash \{\nu : \textit{int} | \nu = \textit{pp} + 1\} <: \{\nu : \textit{int} | \nu > 0\} \quad (2)$$

The `pp` item in the environment is introduced by the let binding. Since `pp` is bound to `this.p.price()`, RFJ sets its type as the type of `this.p.price()`, which is $\{\nu : \textit{int} | \nu > 0\}$.

Refinement for `this` and Override with Co/contra-variance. In RFJ, every method has an implicit `this` parameter with the same type as the enclosing class of this method (e.g., in `Cheese.price()`, `this` has `Cheese` type). We have seen `this` appearing in previous subtyping constraints, but with a trivial refinement `true`. `this` can also be given a non-trivial refinement to ensure that methods are invoked on objects satisfying specific criteria. For instance, `Pizza.sell` includes a refinement of `this` (marked `cyan`), specifying that only the pizza whose `price` is greater than 5 can be sold.

Meanwhile, suppose that a `Crust` can also be sold regardless of its price. This can be achieved by **overriding** the method `sell` in `Crust`, as the example shows. In the overriding method, the refinement of `this` is `true` and thus omitted, making it a supertype of the previous refinement `this.price > 5`, obeying **contra-variance** of parameter types².

Now, consider a property for `Anchovy.price()`: the price is not only positive, but also not less than that of `this.p`. This extra property is marked `olive` in the program. The property makes the new return type a subtype of the old, obeying return type **co-variance**.

General selfification. Checking `Anchovy.price()`'s return type yields this constraint:

$$\textit{this} : \textit{Anchovy}, \textit{pp} : \{\textit{int} | \nu > 0\} \vdash \{\textit{int} | \nu = \textit{pp}\} <: \{\textit{int} | \nu > 0 \& \& \nu \geq \textit{this.p.price}()\} \quad (3)$$

Here, we omit the refinement binder ν and the refinement when it is trivial (i.e., `true`). However, this constraint can not be proved currently, essentially due to the loss of the `this.p.price()` term information in the type of `pp`. Luckily, RFJ's general selfification³ mechanism addresses this by ensuring the persistence of such information. In a nutshell, it works by equating the term being typed to the refinement of its type, giving $\textit{this.p.price}() : \{\nu : \textit{int} | \nu > 0 \& \& \nu = \textit{this.p.price}()\}$, which is also the type of `pp`. The strengthened type of `pp` lets the constraint be proved. With the same technique, we can prove the validity of `MagicAnchovy.price`, which further overrides `Anchovy.price`.

² Strictly speaking, for the type of `this`, we use co-variance for the base type and contra-variance for the refinement, c.f. Section 3.1.

³ We name it *general* to distinguish from the cases like [34], where selfification is only used for variables.

Referring to Methods. Next, we turn to the `remA` methods for removing all anchovies from a pizza. Consider the method `testRemA`, where we assess the correctness of the `remA` implementations. For the assertion in Line 24, RFJ enforces the subtyping constraint below:

$$p1 : \{An|\nu = An(Ch(Cr()))\} \vdash \{Pi|\nu = p1.\text{remA}()\} <: \{Pi|\nu = Ch(Cr())\} \quad (4)$$

We omit `this` from the type environment, which does not affect the meaning of this subtyping constraint. Meanwhile, we abbreviate class names to their initial two letters (e.g., `Ch` represents *Cheese*), and omit the `new` keyword (e.g., `An(Ch(Cr()))` represents *new An(new Ch(new Cr()))*), in order to save space. Proving Constraint (4) demands intricate reasoning about the program’s semantics, particularly the semantics of the `remA` methods. This contrasts with the previous example, where no specific knowledge about the `price` methods is required. In our meta-theoretical calculus, since the logical interpretation is built upon the program semantics, Constraint (4) does not pose a significant challenge. Nevertheless, facilitating its efficient handling within SMT solvers requires a theory about RFJ program semantics, which is discussed in detail in Section 2.2.

Proving with Interfaces. Finally, we consider a more interesting property concerning `price` and `remA`: the price of a pizza should not increase after removing all anchovies. We can express this by appending the following method to `Pizza`: `{bool|this.price()>=this.remA().price()}` `remA_noinc_price(){return true;}`, yielding the subtyping constraint below:

$$this : Pizza \vdash \{bool|\nu = true\} <: \{bool|this.price() \geq this.\text{remA}().\text{price}()\} \quad (5)$$

This property holds in our meta-theoretical calculus. However, it breaks the proof modularity and is not verifiable in the algorithmic verification, even with the theory extended with RFJ program semantics. The mitigation of this challenge is facilitated by another key feature of RFJ: interfaces. We discuss that in detail in the following section.

2.2 Algorithmic Verification

In conventional refinement type systems, subtyping constraints are typically discharged by SMT solvers, which facilitate automated reasoning and significantly reduce implementation efforts. We adapt this methodology by providing a logical encoding of RFJ into a dedicated order-sorted first-order logic, named LFJ. A detailed exposition of LFJ is provided in Section 5. Here, we offer a concise overview of it, drawing upon the examples discussed in Section 2.1.

EUFLIA. After being encoded into LFJ, the subtyping constraints (1), (2), and (3) fall into the theory of Equality, Uninterpreted Functions, and Linear Integer Arithmetic (EUFLIA), a domain widely supported by contemporary SMT solvers [9, 4, 19]. LFJ incorporates EUFLIA for verifying those constraints.

Reasoning about Program Semantics. We have illustrated in Section 2.1 that the verification of Constraint (4) requires knowledge about program semantics. We encode the knowledge with several axioms, which are discussed formally in Section 5.3. Currently, We illustrate them utilizing Constraint (4), which is translated to the LFJ formula shown below:

$$\forall p1 : An, \nu : Pi. p1 = An_{cr}(Ch_{cr}(Cr_{cr}())) \wedge \nu = An_{remA}(p1) \Rightarrow \nu = Ch_{cr}(Cr_{cr}())$$

Here, the An_{cr} , Ch_{cr} , Cr_{cr} functions represent the constructors for the classes `An`, `Ch`, and `Cr`, respectively. An_{remA} represents the *conditional function* composed of the possible *implementations* of `Anchovy.remA`. The characterization of `Anchovy.remA` is shown below:

$$An_{remA}(this) = \begin{cases} Pi_{remA}(An_p(this)) & \text{if } this = An_{cr}(\dots) \\ Pi_{remA}(An_p(this)) & \text{if } this = Ma_{cr}(\dots) \end{cases}$$

This characterization redirects the function application to the implementation, depending on the *class* of *this* (although the implementations are the same). Since the two classes both use `Anchovy.remA`, all the implementations are the logic translation of the method body of `Anchovy.remA`. Here, An_p denotes the access function of the field `p` of the class `An`, while Pi_{remA} is the conditional function for `Pizza.remA`.

Because we know $p1$ is $An_{cr}(Ch_{cr}(Cr_{cr}(\dots)))$, we choose the first branch and deduce $\nu = Pi_{remA}(An_p(An_{cr}(Ch_{cr}(Cr_{cr}(\dots)))))$, which can be then handled by an axiom for An_p :

$$\forall p : Pi. An_p(An_{cr}(p)) = p$$

With this axiom, we can deduce $\nu = Pi_{remA}(Ch_{cr}(Cr_{cr}(\dots)))$. This time, we need to utilize the semantics of Pi_{remA} , and choose the branch for $this = Ch_{cr}(\dots)$, i.e., $Pi_{remA}(this) = Ch_{cr}(Pi_{remA}(Ch_p(this)))$, which lets us deduce $\nu = Ch_{cr}(Pi_{remA}(Ch_p((Ch_{cr}(Cr_{cr}(\dots))))))$. Following the routine we just outlined, we can deduce $\nu = Ch_{cr}(Cr_{cr}(\dots))$ eventually.

Verifying with Interfaces. Even with the knowledge of program semantics, Constraint (5) still can not be verified. In particular, it would be translated to $\forall this : Pi. Pi_{price}(this) \geq Pi_{price}(Pi_{remA}(this))$, whose verification requires induction. In several previous refinement type systems for functional languages [66], induction is supported but is based on pattern matching and recursive functions. In object-oriented languages, pattern-matching constructs are typically not included. Thus, in this paper, we propose to utilize *interface*, a feature that is included in most object-oriented languages, to perform induction. To use interface-based induction in our case, we reimplement `Pizza` as an interface, which decomposes the proof obligation into subclasses implementing `Pizza`. To illustrate, we pick the proof of Constraint (5) for `Anchovy` as an example, shown in Figure 2.

The first thing to note is that, the method body of `Anchovy.remA_noinc_price` is not trivial (e.g., `return true`). This means that an SMT solver would not prove this property automatically. Before we explain the method body, we first give a brief informal proof to help understanding. The assumptions to facilitate the proof are listed below. Once the assumptions are within the proof context, the formula we want ($this.price() \geq this.remA().price()$) can be easily deduced within EUFLIA.

$this.price() \geq this.p.price()$	$(p_1) : \text{a property of Anchovy.price}$
$this.p.price() \geq this.p.remA().price()$	$(p_2) : \text{the induction hypothesis of this.p}$
$this.remA() = this.p.remA()$	$(p_3) : \text{a property of Anchovy.remA}$

The first assumption (property) is about `Anchovy.price`, and we have proved it in Section 2.1, so we can just refer to it as a lemma. In principle, even if we have not proved it, the solver can still automatically prove the property with the given program semantics and use it to prove the formula we want. However, leaving such a property to the solver often increases the searching time for proving the formula. Thus, we prove it as a separate property and introduce it ($p1$ in the body of `remA_noinc_price()`) to the proof context so that the solver can use it directly. The second property is the induction hypothesis of `this.p`

```

25   interface Pizza{
26     {int|v>0} price()
27     Pizza remA()
28     Pizza sell (this.price()>5)
29     {v:bool|this.price()>=this.remA().price()} remA_noinc_price() }
30   class Anchovy implements Pizza{
31     p:Pizza
32     {v:int|v>0 && v>=this.p.price()} price(){return let pp = this.p.price() in pp;}
33     Pizza remA(){return this.p.remA();}
34     {v:bool|this.price()>=this.remA().price()} remA_noinc_price(){
35       return let p1 = this.price() in
36       let p2 = this.p.remA_noinc_price() in true;}}

```

Figure 2 Proving `remA_noinc_price` for `Anchovy`.

($this.p.price() \geq this.p.remA().price()$), and we can utilize it by referring to the method `remA_noinc_price` of `this.p`. The third property asserts that for every `this:Anchovy`, it holds that `this.remA()=this.p.remA()`. This is obvious since no matter whether `this` is an `Anchovy` or a `MagicAnchovy`, calling `remA` on it would resolve to the same method implementation. Like p_1 , this property is also automatically derivable, so we *can* leave it to the SMT solver, or explicitly prove it like we do for p_1 . In particular, p_3 represents a special case where we can add an axiom to the solver, to spare the efforts of automatic search and manual proof. As a result, it is not included in the method body. We will discuss this further in Section 5.3.

We need not prove the same property for `MagicAnchovy`, since it inherits the property from `Anchovy`. Similarly, we can prove other properties such as the fact that no `Anchovy` exists after `remA`, as well as the idempotence of `remA`; all have been included in our test suite.

2.3 Meta-theoretical Arguments

As we have seen in the previous sections: the design of RFJ aims at expressiveness and ease of use. At the same time, this very desirable combination leads to a tricky meta-theory. One major contribution of this paper is to establish the meta-theory rigorously (i.e., in Coq). The detailed development of the meta-theory is given in Section 4. In this section, we briefly overview the soundness theorems, and several challenges in proving them.

2.3.1 Soundness Theorems

Type Soundness. The type system should be able to preclude evaluation from being stuck. Formally, if a closed expression is well-typed ($\emptyset \vdash e : t$), then it would never be stuck in a state where it can not be evaluated and is not a value yet. Since type soundness is typically guaranteed by the base type system (in our case, the FJ type system) already, the core of proving that in refinement type systems is to ensure the additional refinement type mechanisms (e.g., general selfification) do not break the promise of the base type system.

Logical Soundness. The type system should infer only *true* refinements. Formally, if a typing judgment ($\Gamma \vdash e : \{w|p\}$) is made by the type system, the refinement formula p must be *true* whenever the conditions stipulated by Γ are fulfilled. Logical soundness serves as a complement to type soundness, going beyond the guarantee that the evaluations of well-typed programs would never get stuck by also ensuring that the evaluations of such programs adhere to the logical constraints specified by type refinements.

2.3.2 Challenges

Logical Interpretation. Refinements are logical formulas and should be interpreted logically. For example, the subtyping relation between two refinement types ($\Gamma \vdash \{w|p\} <: \{w|q\}$) is defined as the truth of the implication $p \Rightarrow q$ under the assumption set Γ . While our approach to algorithmic verification leverages a translation of RFJ into first-order logic (c.f., Section 2.2), employing this logic directly for defining the logical interpretation can prove both intricate and unwieldy. Rather, we choose to define it *inside the language*, allowing the algorithmic verification approach to serve as an external algorithm that scrutinizes the intrinsic logical interpretation. Nevertheless, articulating a precise logical interpretation is still challenging due to complex typing mechanisms like interfaces and nominal subtyping.

Type Substitution and General Selfification. Different from previous class-based refinement type calculus, RFJ uses type substitution instead of ANF [32] or existential types [34]. This increases the generality and usability (detailed in Section 7). However, this also increases its meta-theoretical complexity, and several nonstandard lemmas about type substitution and typing have to be proved. Similarly, although general selfification increases the precision of the verification by recording term information in refinements, it affects substitution and preservation lemmas intricately and several non-standard properties (e.g., exactness, $\Gamma \vdash e : t$ then $\Gamma \vdash e : \text{self}(t, e)$) of it have to be proved for proving those lemmas.

First Order Functions. The logical relation technique [62] is frequently employed in prior research [8, 29] to establish the logical soundness theorem. However, this technique can not be applied to RFJ, since RFJ is a first-order language without explicit function abstraction but with recursive method definition. Thus, we do not have a strong enough induction principle about methods when performing induction on typing. This challenge is not unique to us and has been encountered in previous studies [64, 55]. However, the workaround adopted by these studies, which essentially inlines methods at call-sites, is incompatible with RFJ, since that requires particular type structures supporting the strong normalisation of derivation reduction, a property that RFJ lacks.

3 Declarative Calculus: RFJ

This section outlines the syntax, semantics, and typing rules of RFJ, built upon the classical calculus FJ extended with primitive data types (integers and booleans) and let bindings. The FJ parts follow closely the classical textbook presentation [52]. To delineate the extensions unique to RFJ, we highlight the extended features in *gray background*.

3.1 Syntax and Lookup Functions

The syntax of RFJ is depicted on the left side of Figure 3. The metavariables C , D , and E range over class names; f and g range over field names; m ranges over method names; x ranges over parameter names; ν ranges over refinement binder names. We also use n to range over integers, and b to range over booleans (i.e., *true* and *false*). In a nutshell, RFJ extends FJ by refinement types, interfaces, and the \top type, each highlighted in dark gray to distinguish the enhancements. We have discussed refinement types and interfaces extensively. For the \top type, it is introduced mainly to characterize the equality between any two values, not just values of the same type. Compared to strictly monomorphic equality which demands type uniformity for comparands, \top -typed equality is closer to the actual Java equality [28] and the equality used in order-sorted logics [57].

Syntax	Sub-nominal	Base-subtyping	Subtyping
$C ::= \text{class definitions:}$	$N_1 <:_n N_2$	$w <:_b u$	$\Gamma \vdash s <: t$
$\text{class } C \text{ extends } D \text{ implements } \bar{I} \{\bar{t} \bar{f}; K \bar{M}\}$			$N <:_n N$
$\mathcal{I} ::= \text{interface } I\{\bar{Q}\} \text{ interface } \text{Defs.}$			
$K ::= C(\bar{t} \bar{f}) \{\text{super}(\bar{f}); \text{this.}\bar{f} = \bar{f};\}$			
$Q ::= t m(p, t x) \quad \text{method } \text{Decls.}$			
$M ::= Q\{\text{return } e;\}$			
$e, p, q ::= \text{terms:}$			
$x \quad \text{variable}$			
$e.f \quad \text{field access}$			
$e.m(e) \quad \text{method invocation}$			
$\text{new } C(\bar{e}) \quad \text{instance creation}$			
$n \quad \text{integer}$			
$b \quad \text{boolean}$			
$\neg e \quad \text{unary operation}$			
$e \oplus e \quad \text{binary operation}$			
$\text{let } x = e \text{ in } e \quad \text{let binding}$			
$\oplus ::= = \vee \wedge \quad \text{binary operators}$			
$v ::= n b \text{new } C(\bar{v}) \quad \text{values}$			
$N ::= C \mid I \quad \text{nominal types}$			
$w, u ::= \top \mid \text{int} \mid \text{bool} \mid N \quad \text{base types}$			
$s, t, r ::= \{\nu : w p\} \quad \text{refinement types}$			
	$N_1 <:_n N_2$	$w <:_b u$	$\Gamma \vdash s <: t$
			$N <:_n N$
	$\frac{N_1 <:_n N_2 \quad N_2 <:_n N_3}{N_1 <:_n N_3}$		
			$\frac{CT(C) = \text{class } C \text{ extends } D \dots \{\dots\}}{C <:_n D}$
		$\frac{CT(C) = \text{class } C \dots \text{implements } \bar{I}\{\dots\}}{C <:_n I_i}$	
			$w <:_b \top$
			$\text{int} <:_b \text{int}$
			$\text{bool} <:_b \text{bool}$
	$\frac{N_1 <:_n N_2}{N_1 <:_b N_2}$		
		$\frac{w <:_b u \quad \Gamma, \nu : w \models p \Rightarrow q}{\Gamma \vdash \{\nu : w p\} <: \{\nu : u q\}}$	
			(R-SUBTYPING)

Figure 3 Syntax and subtyping.

Besides the extension, RFJ simplifies FJ in two aspects, widely adopted in prior studies [55, 10, 42, 27]. First, casts are not included since they complicate the calculus and are orthogonal with refinement types, the focus of this work. Second, a single parameter is used instead of an arbitrary number of parameters. However, this does not impact the expressiveness of RFJ, because empty parameters can be modeled by a single parameter that is not referred to in the method body, while multiple parameters can be modeled by declaring a class containing those parameters and using it as a single parameter.

Finally, we introduce several remarkable notations. Firstly, note that we use e , p , and q to range over RFJ terms. The latter symbols (p and q) are used to range over RFJ terms that have bool type (also called *formulas*). Secondly, we use two shorthands for refinement types: we omit the binders declaration in $\{\nu : w|p\}$ when the binder is just ν (a reserved name), and we short $\{w|p\}$ as w when p is *true*.

Subtyping. The right half of Figure 3 explicates RFJ’s subtyping relations, featuring sub-nominal ($<:_n$), subtyping amongst base types ($<:_b$), and refinement subtyping ($<:$). The sub-nominal relation is a straightforward extension of FJ’s subclassing to account for interface types. The base subtyping relation is also standard. Refinement subtyping combines base subtyping and logical implication (defined in Section 3.3). It is parameterized by type environments with the usual construction, which is used for logical implication. Note that

39:10 Formalizing, Mechanizing, and Verifying Class-Based Refinement Types

when checking logical implication, we use the type of the sub-base-type (w) instead of the super-base-type (u) for ν , to make refinement subtyping transitive. In the remaining of this paper, we short refinement subtyping as subtyping when there is no ambiguity.

Field lookup $fields(C) = \bar{t} \bar{f}$ $fields(Object) = \bullet$ $\frac{CT(C) = class\ C\ exds\ D\ imps\ \bar{I}\{\bar{t}\ \bar{f};\ K\ \bar{M}\} \quad fields(D) = \bar{s}\ \bar{g}}{fields(C) = \bar{s}\ \bar{g}, \bar{t}\ \bar{f}}$ C-method-type $mtype(m, C) = p \rightarrow x : t \rightarrow r$ $\frac{CT(C) = class\ C\ exds\ D\ imps\ \bar{I}\{\bar{t}\ \bar{f};\ K\ \bar{M}\} \quad r\ m\ (\underline{p},\ t\ x)\ \{return\ e;\} \in \bar{M}}{mtype(m, C) = \underline{p} \rightarrow x : t \rightarrow r}$ $\frac{CT(C) = class\ C\ exds\ D\ imps\ \bar{I}\{\bar{t}\ \bar{f};\ K\ \bar{M}\} \quad m\ is\ not\ defined\ in\ \bar{M}}{mtype(m, C) = mtype(m, D)}$ C-method-body $mbody(m, C) = (x, e)$ $\frac{CT(C) = class\ C\ exds\ D\ imps\ \bar{I}\{\bar{t}\ \bar{f};\ K\ \bar{M}\} \quad r\ m\ (\underline{p},\ t\ x)\ \{return\ e;\} \in \bar{M}}{mbody(m, C) = (x, e)}$ $\frac{CT(C) = class\ C\ exds\ D\ imps\ \bar{I}\{\bar{t}\ \bar{f};\ K\ \bar{M}\} \quad m\ is\ not\ defined\ in\ \bar{M}}{mbody(m, C) = mbody(m, D)}$	Override $\frac{override(m, C, D, p \rightarrow x : t \rightarrow r) \quad mtype(m, D) = q \rightarrow x : t' \rightarrow r'}{\begin{array}{l} \Rightarrow (\emptyset \vdash \{C q\} <: \{C p\}) \\ \emptyset, this : \{C q\} \vdash t' <: t \\ \emptyset, this : \{C q\}, x : t' \vdash r <: r' \end{array}} \overline{override(m, C, D, p \rightarrow x : t \rightarrow r)}$ I-method-type $\frac{mtypei(m, I) = p \rightarrow x : t \rightarrow r \quad IT(I) = interface\ I\{\bar{Q}\} \quad r\ m\ (\underline{p},\ t\ x) \in \bar{Q}}{mtypei(m, I) = p \rightarrow x : t \rightarrow r}$ Implement $\frac{implement(m, C, I, p \rightarrow x : t \rightarrow r) \quad mtype(m, C) = q \rightarrow x : t' \rightarrow r' \quad \begin{array}{l} \emptyset \vdash \{C p\} <: \{C q\} \\ \emptyset, this : \{C p\} \vdash t <: t' \\ \emptyset, this : \{C p\}, x : t \vdash r' <: r \end{array}}{\overline{implement(m, C, I, p \rightarrow x : t \rightarrow r)}}$ Interface implemented $\frac{IT(I) = interface\ I\{\overline{r\ m(p,\ t\ x)}\} \quad implement(m, C, I, \overline{p \rightarrow x : t \rightarrow r})}{C \triangleright I}$
--	---

Figure 4 Auxiliary definitions.

Auxiliary Definitions. The lookup functions, override relation, and *interface implemented* relation are shown in Figure 4. The lookup functions should be pretty self-explanatory, only to note that although we use \rightarrow in *mtype* and *mtypei*, it is not *arrow type* constructor, but an intuitive type signature representation, as in original FJ [11]. We explain the override and *interface implemented* relations subsequently.

In RFJ, the criterion for valid method override differs from FJ’s strict type matching, utilizing co/contra-variance instead. This is encapsulated by the override relation ($override(m, C, D, p \rightarrow x : t \rightarrow r)$), which ensures the class C properly overrides the method m of the class D (if m does exists in D), encoding three constraints:

1. For **this**, we have co-variance in the base type (the base type is C , which is a subtype of D) and contra-variance in the refinement (as the $\emptyset \vdash \{C|q\} <: \{C|p\}$ states). Using covariance for the base type is widely known as a seminal work [11] on method overriding has pointed out: the parameters that determine the selection must be co-variantly overridden (i.e., have a lesser type). However, since method selection relies solely on the base type (note that *mbody* considers the class but disregards refinement), we must require the refinement to be more general (contra-variant) to ensure compatibility.

2. The contra-variance on the parameter type and co-variance on the return type (ignore the subtyping context for now) follow the function subtyping principle [52].
3. Since the parameter type refinement may refer to `this`, while the return type refinement may refer to `this` and the parameter, their co/contra-variance must be assessed under a type environment with those variables, as the definition shows. Here, note that we opt for “narrower” subtype contexts: we assess contra-variance ($t' <: t$) within the context of $\{C|q\}$ rather than $\{C|p\}$, and likewise for co-variance ($r <: r'$), within $\{C|q\}$ and t' . This decision renders the overriding rule more permissive: subtyping in a narrower context is easier to satisfy, as the narrowing property of subtyping shows (c.f., Section 4.3.1).

Finally, note that we simplify the presentation by assuming identical parameter names (x); otherwise, they should be renamed to a fresh variable for checking return type co-variance.

For valid interface implementations, $C \triangleright I$ confirms that a class properly implements all methods declared in the interface. The implementation relation ($implement(m, C, I, p \rightarrow x : t \rightarrow r)$) is a dual of the override relation, ensuring that the method m of interface I with type signature $p \rightarrow x : t \rightarrow r$, is overridden in the class C . Note that the formalization is different from override in that, *override* only requires the subtyping constraints to hold *if the method m exist*, while *implement* requires the method m does exist, and satisfies the subtyping constraints.

3.2 Operational Semantics

Evaluation	$e \rightsquigarrow e'$	
	$\boxed{e \rightsquigarrow e'}$	
	$\frac{fields(C) = \bar{t} \bar{f}}{(new\ C(\bar{v})).f_i \rightsquigarrow v_i}$	
	$\frac{mbody(m, C) = (x, e_0)}{(new\ C(\bar{v})).m(v) \rightsquigarrow [this \mapsto (new\ C(\bar{v})); x \mapsto v]e_0}$	
	$\frac{e_0 \rightsquigarrow e'_0}{e_0.f \rightsquigarrow e'_0.f}$	
	$\frac{e_0 \rightsquigarrow e'_0}{e_0.m(e) \rightsquigarrow e'_0.m(e)}$	
	$\frac{e \rightsquigarrow e'}{v_0.m(e) \rightsquigarrow v_0.m(e')}$	
	$\frac{e_i \rightsquigarrow e'_i}{new\ C(\bar{v}, e_i, \bar{e}) \rightsquigarrow new\ C(\bar{v}, e'_i, \bar{e})}$	
		$\frac{e \rightsquigarrow e'}{\neg e \rightsquigarrow \neg e'}$
		$\frac{\neg b \rightsquigarrow \neg_p(b)}{e_0 \rightsquigarrow e'_0}$
		$\frac{e_0 \oplus e \rightsquigarrow e'_0 \oplus e}{e \rightsquigarrow e'}$
		$\frac{v_0 \oplus e \rightsquigarrow v_0 \oplus e' \oplus ok\ v_0\ v_1}{v_0 \oplus v_1 \rightsquigarrow \oplus_p(v_0, v_1)}$
		$\frac{e_0 \rightsquigarrow e'_0}{let\ x = e_0\ in\ e \rightsquigarrow let\ x = e'_0\ in\ e}$
		$\frac{let\ x = v_0\ in\ e \rightsquigarrow [x \mapsto v_0]e}{\boxed{\oplus\ ok\ v_0\ v_1}}$
		$\frac{\wedge\ ok\ b_0\ b_1}{\vee\ ok\ b_0\ b_1}$
		$\frac{}{= ok\ v_0\ v_1}$
		$\text{Valid binary operation}$

Figure 5 Small-step semantics of RFJ.

Now, we present the operational semantics of RFJ. We first present the small-step semantics, defined in Figure 5. The semantics aligns with that of FJ⁴, diverging only to accommodate the integration of new constructs – specifically, primitive operations and let bindings. The standard semantics of the boolean operations – including negation, conjunction, and disjunction, are preserved. The only thing worth noting is the equality operation, which is defined for every pair of values. RFJ equality is defined as the syntactic equality (i.e., we view values as finite term trees [22]: two values are equal *iff* their corresponding trees are identical).

Multi-step and Big-step Semantics. We define the multi-step semantics ($e \rightsquigarrow^* e'$) as the transitive closure of the small-step semantics, used for type soundness and logical truth later.

Despite being directly derivable from small-step semantics, multi-step semantics do not provide a convenient induction principle, which makes the related proof intricate. To mitigate this, we introduce big-step semantics and prove its coincidence with the multi-step semantics terminating with a value (i.e., $e \Downarrow v$ *iff* $e \rightsquigarrow^* v$). The big-step semantics mirrors the small-step semantics, and we omit its formal definition from this paper. We defer its comprehensive exposition to the accompanying Coq development.

3.3 Logical Interpretation

Figure 6 defines the logical notations, which are used in the refinement subtyping relation and logical soundness theorem. The definitions make use of closing substitutions, i.e., partial mappings from variables to values. The application of a closing substitution θ to a term e is defined as the function $\theta(e)$, which simply substitutes each variable-value pair sequentially. We also lift $\theta(\cdot)$ to refinement types: $\theta(\{\nu : w|p\}) = \{\nu : w|\theta(p)\}$.

Logical Truth and Entailment. The core of our logical interpretation is the logical truth relation, which means that the logical formula evaluates to *true* under the given interpretation (i.e., RFJ operational semantics). Note that this relation is defined only for closed formulas (i.e., *sentences*), and a closing substitution is applied whenever this relation is checked.

With the logical truth relation in hand, we can define the logical entailment relation ($\Gamma \models p$), which signifies the truth of a formula p under the type and logical constraints encoded within Γ . It requires that for every closing substitution that satisfies Γ (formally defined later), the closing substitution must also satisfy the formula p (i.e., make it a truth). Similarly, we define the logical implication relation ($\Gamma \models p \Rightarrow q$), by requiring all closing substitution that satisfies Γ and p also satisfies q .

The logical implication relation is used for defining the subtyping relation (c.f., Section 3.1). To illustrate, we revisit the subtype constraint (4) presented in Section 2.1, which imposes the following constraint by the definition of subtyping and logical implication:

$$\forall \theta \in [\Gamma]. \text{ if } \models \theta(\nu = p1.\text{rem}A()) \text{ then } \models \theta(\nu = \text{new } Ch(\text{new } Cr()))$$

where $\Gamma = p1 : \{An|\nu = \text{new } An(\text{new } Ch(\text{new } Cr()))\}$, $\nu : Pi$. There are infinite closing substitutions satisfying Γ . In particular, $p1$ can only be $\text{new } An(\text{new } Ch(\text{new } Cr()))$, but ν can be any *Pizza*, since any *Pizza* v satisfies $v \in [Pi]$. However, there is only one closing substitution that also satisfies the *if* condition ($\models \theta(\nu = p1.\text{rem}A())$), i.e., the one whose ν is $\text{new } Ch(\text{new } Cr())$. This closing substitution also satisfies the *then* condition. Thus, Constraint (4) holds under the logical interpretation.

⁴ To be specific, we align with the semantics in the textbook presentation [52], which diverges from the nondeterministic beta-reduction semantics in the original paper [30].

$\theta ::= \theta, x : v \mid \emptyset$	Environment Denotation	$\theta \in \llbracket \Gamma \rrbracket$
Closing Substitution		
$\emptyset(e) = e$		$\emptyset \in \llbracket \emptyset \rrbracket$
$(\theta, x : v)(e) = [x \mapsto v]\theta(e)$		$\frac{v \in \llbracket \theta(t) \rrbracket \quad \theta \in \llbracket \Gamma \rrbracket}{\theta, x : v \in \llbracket \Gamma, x : t \rrbracket}$
Logical Truth	$\models p$	Type Denotation
		$v \in \llbracket t \rrbracket$
$\frac{p \rightsquigarrow^* \text{true}}{\models p}$		$\frac{\models [\nu \mapsto n]p \quad n \in \llbracket \{\nu : \text{int} p\} \rrbracket}{n \in \llbracket \{\nu : \text{int} p\} \rrbracket}$
Logical Entailment	$\Gamma \models p$	(DENINT)
		$\frac{\models [\nu \mapsto b]p}{b \in \llbracket \{\nu : \text{bool} p\} \rrbracket}$
		(DENBOOL)
Logical Implication	$\Gamma \models p \Rightarrow q$	
		$\frac{\models [\nu \mapsto \text{new } C(\bar{v})]p \quad \text{fields}(C) = \bar{t} \quad \bar{v} \in \llbracket \text{this} \mapsto \text{new } C(\bar{v}) \mid t \rrbracket}{\text{new } C(\bar{v}) \in \llbracket \{\nu : C p\} \rrbracket}$
		(DENCLASS)
		$\frac{w <:_b u \quad v \in \llbracket \{\nu : w p\} \rrbracket}{v \in \llbracket \{\nu : u p\} \rrbracket}$
		(UPCAST)

Figure 6 Logical interpretation of RFJ.

Type and Environment Denotation. Now, we formally define what is meant by “a substitution satisfies a type environment.” This relation is defined by the environment denotation relation $\theta \in \llbracket \Gamma \rrbracket$, which is a natural lift of the type denotation relation ($v \in \llbracket t \rrbracket$), determining if a value is denoted by a type. Type denotation is defined by casing on the structure of the value, with an additional *upcast* rule for upcasting the base type. Basically, type denotation relation ($v \in \llbracket \{u|p\} \rrbracket$) encapsulates two facets: the value v belongs to the base type u , and it satisfies the refinement p . DENCLASS additionally requires the denotation for the fields of the class, to justify the nominal nature of class types.

3.4 Typing

In this section, we define the typing relations in RFJ, as shown in Figure 7. We first define the term typing, depending on the type well-formedness relation, which in turn depends on the FJ term typing. After the term typing is defined, we define the method typing ($M \text{ ok in } C$), class typing ($C \text{ ok}$), and interface typing ($I \text{ ok}$).

Well-formedness. For a refinement type $\{\nu : w|p\}$ to be deemed well-formed under environment Γ , denoted as $\Gamma \vdash_w \{\nu : w|p\}$, the refinement p must have *bool* type under the type environment. In the definition, \vdash_F is the FJ term typing relation, which is used to check if the refinement does have *bool* type. Note that we can not use the RFJ term typing here, since it depends on the type well-formedness relation. We do not define the FJ term typing separately. It is a standard textbook relation [52] and can be obtained by removing

Type well-formedness	$\boxed{\Gamma \vdash_w t}$	$\frac{\Gamma \vdash e_0 : s_0 \quad \Gamma, x : s_0 \vdash e : t \quad \Gamma \vdash_w t}{\Gamma \vdash \text{let } x = e_0 \text{ in } e : \boxed{\text{self}(\boxed{t}, \boxed{\text{let } x = e_0 \text{ in } e})}}$ (T-LET)
	$\frac{[\Gamma], \nu : w \vdash_F p : \text{bool}}{\Gamma \vdash_w \{\nu : w p\}}$	$\frac{\neg_t \doteq x : t_0 \rightarrow r \quad \Gamma \vdash s_0 <: t_0}{\Gamma \vdash \neg e_0 : \boxed{[x \mapsto e_0]} r}$ (T-UNOP)
RFJ Typing	$\boxed{\Gamma \vdash e : t}$ $\text{self}(\{\nu : w p\}, e) = \{\nu : w p \wedge \nu = e\}$	$\frac{x : t \in \Gamma \quad \boxed{\Gamma \vdash_w t}}{\Gamma \vdash x : \boxed{\text{self}(\boxed{t}, \boxed{x})}}$ (T-VAR)
		$\frac{x : t \in \Gamma \quad \Gamma \vdash_w t}{\Gamma \vdash n : \boxed{\text{int} \mid \nu = n}}$ (T-INT)
		$\frac{x : t \in \Gamma \quad \Gamma \vdash_w t}{\Gamma \vdash b : \boxed{\text{bool} \mid \nu = b}}$ (T-BOOL)
		$\frac{\Gamma \vdash e_0 : \{\boxed{C_0 \mid p}\} \quad \text{fields}(C_0) = \bar{t} \bar{f}}{\Gamma \vdash e_0.f_i : \boxed{\text{self}([\text{this} \mapsto e_0] t_i, \boxed{e_0.f_i})}}$ (T-FIELD)
		$\frac{\Gamma \vdash e_0 : \{\boxed{C_0 \mid p}\} \quad \Gamma \vdash \{C_0 p\} <: \{C_0 q\} \quad \Gamma \vdash e : s \quad \Gamma \vdash s <: \boxed{[\text{this} \mapsto e_0] t}}{\Gamma \vdash e_0.m(e) : \boxed{\text{self}([\text{this} \mapsto e_0; x \mapsto e] r, \boxed{e_0.m(e)})}}$ (T-INVOK)
		$\frac{\Gamma \vdash e_0 : \{I_0 p\} \quad \Gamma \vdash \{I_0 p\} <: \{I_0 q\} \quad \Gamma \vdash e : s \quad \Gamma \vdash s <: \boxed{[\text{this} \mapsto e_0] t}}{\Gamma \vdash e_0.m(e) : \boxed{\text{self}([\text{this} \mapsto e_0; x \mapsto e] r, \boxed{e_0.m(e)})}}$ (T-INVOKI)
		$\frac{\Gamma \vdash \bar{e} : \bar{s} \quad \Gamma \vdash \bar{s} <: \boxed{[\text{this} \mapsto \text{new } C(\bar{e})] t}}{\Gamma \vdash \text{new } C(\bar{e}) : \boxed{\text{self}(\boxed{C}, \boxed{\text{new } C(\bar{e})})}}$ (T-NEW)
		$\frac{\Gamma \vdash e_0 : s_0 \quad \Gamma, x : s_0 \vdash e : t \quad \Gamma \vdash_w t}{\Gamma \vdash \text{let } e_0 : \boxed{[x \mapsto e_0]} r}$ (T-BINOP)
		$\frac{\Gamma \vdash e : s \quad \Gamma \vdash s <: \boxed{[x \mapsto e_0] t}}{\Gamma \vdash e_0 \oplus e : \boxed{[x \mapsto e_0; y \mapsto e] r}}$ (T-BINOP)
		$\frac{\Gamma \vdash e : s \quad \Gamma \vdash s <: t \quad \boxed{\Gamma \vdash_w t}}{\Gamma \vdash e : t}$ (T-SUB)
method typing		$\frac{M \text{ ok in } C}{CT(C) = \text{class } C \text{ exds } D \text{ imps } \bar{I}\{\dots\}}$ $\text{this} : \{C p\}, x : t \vdash e_0 : r$ $\text{override}(m, C, D, p \rightarrow x : t \rightarrow r)$ $\emptyset \vdash_w \{C p\} \quad \text{this} : \{C p\} \vdash_w t$ $\text{this} : \{C p\}, x : t \vdash_w r$
class typing		$\frac{r m(p, t x)\{\text{return } e_0; \} \text{ ok in } C}{K = C(\bar{s} \bar{g}, \bar{t} \bar{f})\{\text{super}(\bar{g}); \text{this}.\bar{f} = \bar{f}; \}}$ $\text{fields}(D) = \bar{s} \bar{g} \quad \bar{M} \text{ ok in } C$ $\emptyset, \text{this} : C \vdash_w \bar{t} \quad C \triangleright \bar{I}$
interface method typing		$\frac{Q \text{ ok in } I}{\emptyset \vdash_w \{I p\} \quad \text{this} : \{I p\} \vdash_w t}$ $\text{this} : \{I p\}, x : t \vdash_w r$
interface ok		$\frac{r m(p, t x) \text{ ok in } I}{\overline{Q} \text{ ok in } I}$ $\overline{Q} \text{ ok in } I$
		$\text{interface } I\{\overline{Q}\} \text{ ok}$

Figure 7 Typing relations of RFJ.

the `gray` parts of RFJ typing. Since the FJ term typing is only defined for base types and base type environments, we must use an erase function ($\lfloor \cdot \rfloor$) to convert refinement type environments to base type environments. The erase function is naturally lifted from the erase function of refinement types (i.e., $\lfloor \{\nu : w|p\} \rfloor = w$).

Based on the type well-formedness, we define the well-formedness of type environment:

$$(1) \vdash_w \emptyset \quad (2) \vdash_w \Gamma, \Gamma \vdash_w t, x \notin \Gamma \implies \vdash_w \Gamma, x : t$$

which simply asserts that all types are well-formed and all variables are unique.

Term Typing. RFJ term typing is an extension of FJ term typing, replacing base types with refinement types and using refinement subtyping for subtyping. Notably, RFJ term typing utilizes an explicit subsumption rule (T-SUB), which deviates from the implicit algorithmic subtyping commonly attributed to FJ. This deviation is not borne from necessity but is rather a methodological choice, aimed at simplifying the meta-theoretical development.

The types of primitive operations (used in T-UNOP and T-BINOP) follow their semantics:

$$\begin{aligned}\neg_t &\doteq x : \text{bool} \rightarrow \{\text{bool} | v = \neg x\} \\ \wedge_t &\doteq x : \text{bool} \rightarrow y : \text{bool} \rightarrow \{\text{bool} | v = x \wedge y\} \\ \vee_t &\doteq x : \text{bool} \rightarrow y : \text{bool} \rightarrow \{\text{bool} | v = x \vee y\} \\ =_t &\doteq x : \top \rightarrow y : \top \rightarrow \{\text{bool} | v = x = y\}\end{aligned}$$

RFJ typing utilizes several mechanisms absent in FJ typing, i.e., well-formedness checking, type substitution, and general selfification. We briefly discuss those non-standard mechanisms.

1. Well-formedness checking. Three rules (T-VAR, T-LET, and T-SUB) include type well-formedness checking in their premises, guaranteeing the inference of only well-formed types, which is required to establish various lemmas (e.g., the structural properties).
2. Type substitution. Refinement types can refer to visible variables. For example, the type of a field `f` can be $\{\nu : \text{int} | \nu = \text{this}.h\}$, specifying it equal to the `h` field of the object. For those refinements to refer to proper variables, we must substitute these references with actual terms during typing. Continuing the example, suppose we are typing `a.f`, the type should be updated to $\{\nu : \text{int} | \nu = a.h\}$, by substituting `this` to `a`, as T-FIELD rule shows.
3. General selfification. Each rule except the subsumption rule and the rules for primitives (T-INT, T-BOOL, T-UNOP and T-BINOP) is companioned with a selfification operation (*self*), ensuring the terms are always recorded in their types. selfification is not required for subsumption, as it is performed in prior derivations, and primitive rules inherently equate terms in their types (e.g., T-INT assigns $\{\text{int} | \nu = 2\}$ to 2).

Method, Class Typing and Interface Typing. The method, class, and interface typings are relations to identifying valid methods, classes, and interfaces. RFJ's approach to these typings closely mirrors that of FJ, with the addition of well-formedness checks for method and field types. Additionally, the class typing judgment is extended with a checking $C \triangleright \bar{I}$ that ensures the interfaces are properly implemented.

Termination. Finally, we address one tricky issue in typing: termination. As a Turing-complete language, the well-typedness of RFJ terms does not ensure the termination of its evaluation. However, non-terminating evaluations can lead to unsound refinements. For instance, $\emptyset \vdash \text{new } C().m() : \{\text{bool} | 0 = 1\}$ is derivable, where `C.m` is defined as `bool m(){return this.m();}`. Consequently, our logical soundness theorem is strictly applicable to terms that are both well-typed and **terminating** (defined below). In practice, a termination checker should be equipped to ensure the termination where logical soundness is concerned.

$$\frac{\forall \theta \in \llbracket \Gamma \rrbracket. \theta(e) \rightsquigarrow^* v}{\Gamma \downarrow e} \text{ terminating}$$

Main Theorems. The following theorems link typing to semantics and logical entailment.

- **Theorem 1** (Type Soundness). *If $\emptyset \vdash e : t$ and $e \rightsquigarrow^* e'$, then e' is a value or $\exists e''. e' \rightsquigarrow e''$.*
- **Theorem 2** (Logical Soundness). *If $\Gamma \vdash e : \{\nu : w | p\}$, $\vdash_w \Gamma$, and $\Gamma \downarrow e$, then $\Gamma \models [\nu \mapsto e]p$.*

The major steps to establish those theorems are given in the next section.

4 Meta-theoretical Results

We argue the proposed system possesses type soundness and logical soundness. The proof of type soundness follows the “Type Soundness = Preservation + Progress” approach [70]. The approach to logical soundness is different from that of previous refinement type systems, as their approach does not apply to RFJ (c.f., Section 2.3). Our proof approach can be summarized as “Logical Soundness = Preservation + Typing Denotation + Closing Substitution.” We give an overview of the critical lemmas and theorems used in the proof and the dependency relation in Figure 8. In the remainder of this section, we provide a brief overview of the proof. For a detailed exposition, please refer to the Coq development.

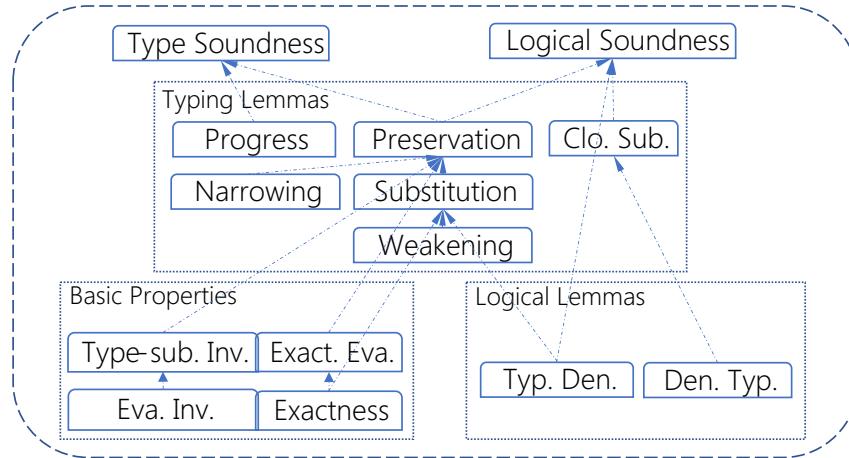


Figure 8 Proof Overview. Arrows signify the dependencies among lemmas and theorems.

4.1 Basic Properties

► **Lemma 3** (Evaluation Invariant). *If $e \rightsquigarrow e'$, then $[x \mapsto e]p \rightsquigarrow^* v \Leftrightarrow [x \mapsto e']p \rightsquigarrow^* v$.*

This lemma asserts that evaluation remains unaffected by the substitution with pre-or-post-evaluation terms, as the next lemma shows. Since the multi-step evaluation (\rightsquigarrow^*) does not give a very useful induction principle, we first prove this lemma using the big-step semantics, then link the lemma back to multi-step semantics via the correspondence between big-step and multi-step semantics (i.e., $e \Downarrow v \Leftrightarrow e \rightsquigarrow^* v$).

► **Lemma 4** (Type-substitution Invariant). *If $e \rightsquigarrow e'$, then $\Gamma \vdash [x \mapsto e]t <: [x \mapsto e']t$ and $\Gamma \vdash [x \mapsto e']t <: [x \mapsto e]t$.*

This lemma states the coherence of types under substitution with pre-or-post-evaluation terms. This lemma is important to prove the preservation lemma. Since subtyping relies eventually on evaluation, the primary challenge of proving this lemma hinges on Lemma 3.

► **Lemma 5** (Exactness). *If $\Gamma \vdash e : t$ then $\Gamma \vdash e : \text{self}(t, e)$.*

This lemma states what we mean by “term information is always recorded”: for any well-typed term e , we can always construct a typing where the term is selfified (recorded in the type). Apart from being used for Lemma 6, this lemma is important for the substitution lemma (Lemma 10).

► **Lemma 6** (Exactness Evaluation). *If $e \rightsquigarrow e'$ and $\Gamma \vdash e' : t$ then $\Gamma \vdash e' : \text{self}(t, e)$.*

This lemma ensures the term after evaluation (e') can have the type selfified with the term before evaluation (e), which is often needed to prove the preservation of typing throughout evaluation steps. This lemma requires the exactness lemma, as shown above.

4.2 Logical Lemmas

► **Lemma 7** (Typing Denotation). *If $\Gamma \vdash v : t$, then $\forall \theta \in \llbracket \Gamma \rrbracket. v \in \llbracket \theta(t) \rrbracket$.*

This lemma states that typing implies denotation. It can be proved by induction on the typing judgment. This lemma is important for the substitution lemma (Lemma 10) and is a milestone for logical soundness, as we discuss in Section 4.5.

► **Lemma 8** (Denotation Typing). *If $v \in \llbracket t \rrbracket$ and $\emptyset \vdash_w t$, then $\emptyset \vdash v : t$.*

This lemma states that denotation implies typing, which is crucial for Lemma 14. The basic proof idea is to first construct a “ground type” for v : $\emptyset \vdash v : \{\nu : w | \nu = v\}$, where w is the *inherent* base type of the value (*int* for n , *bool* for b , and C for *new C(...)*), and then link the “ground type” to t by $\emptyset \vdash \{\nu : w | \nu = v\} <: t$, which holds due to $v \in \llbracket t \rrbracket$.

4.3 Typing Lemmas

4.3.1 Structural Lemmas for Typing

As usual, we establish structural properties (weakening, narrowing and substitution) for RFJ typing. Since typing relies on subtyping which in turn, relies on logical implication, we need those structural properties for subtyping and logical implication, too.

► **Lemma 9** (Narrowing). *for any variable x not in Γ and Γ' :*

1. *If $\Gamma, x : r, \Gamma' \vdash p \Rightarrow q$ and $\Gamma \vdash r' <: r$, then $\Gamma, x : r', \Gamma' \vdash p \Rightarrow q$.*
2. *If $\Gamma, x : r, \Gamma' \vdash s <: t$ and $\Gamma \vdash r' <: r$, then $\Gamma, x : r', \Gamma' \vdash s <: t$.*
3. *If $\Gamma, x : r, \Gamma' \vdash e : t$ and $\Gamma \vdash r' <: r$, then $\Gamma, x : r', \Gamma' \vdash e : t$.*

The first narrowing lemma can be proved by observing that a denotation θ' of $\Gamma, x : r', \Gamma'$ is always a denotation of $\Gamma, x : r, \Gamma'$. Using the first lemma, the remaining two are easy.

► **Lemma 10** (Substitution). *for any distinct variables x and y not in Γ and Γ' :*

1. *If $\Gamma, x : r_x, y : r_y, \Gamma' \vdash p \Rightarrow q$ and $\Gamma \vdash v_x : r_x$, $\Gamma \vdash v_y : [x \mapsto v_x]r_y$, then $\Gamma, [x \mapsto v_x; y \mapsto v_y]\Gamma' \vdash [x \mapsto v_x; y \mapsto v_y]p \Rightarrow [x \mapsto v_x; y \mapsto v_y]q$.*
2. *If $\Gamma, x : r, y : r_y, \Gamma' \vdash s <: t$ and $\Gamma \vdash v_x : r_x$, $\Gamma \vdash v_y : [x \mapsto v_x]r_y$, then $\Gamma, [x \mapsto v_x; y \mapsto v_y]\Gamma' \vdash [x \mapsto v_x; y \mapsto v_y]s <: [x \mapsto v_x; y \mapsto v_y]t$.*
3. *If $\Gamma, x : r, y : r_y, \Gamma' \vdash e : t$ and $\Gamma \vdash v_x : r_x$, $\Gamma \vdash v_y : [x \mapsto v_x]r_y$, then $\Gamma, [x \mapsto v_x; y \mapsto v_y]\Gamma' \vdash [x \mapsto v_x; y \mapsto v_y]e : [x \mapsto v_x; y \mapsto v_y]t$.*

Since RFJ has double substitution operations in method invocation (we must substitute for *this* and the parameter), we need double substitution lemmas. The first substitution lemma follows from the observation that a denotation of $\Gamma, x : r_x, y : r_y, \Gamma'$ can be constructed from a denotation of $\Gamma, [x \mapsto v_x; y \mapsto v_y]\Gamma'$ by adding $x : v_x$ and $y : v_y$. The core step of this construction is to prove that v_x is indeed a denotation of r_x and v_y is indeed a denotation of $[x \mapsto v_x]r_y$, utilizing Lemma 7. Using the first lemma, the second lemma is easy. The third lemma can be proved by induction on typing. The T-VAR case requires the exactness lemma (Lemma 5) and weakening lemma (shown below). The other cases are easy.

► **Lemma 11** (Weakening). *for any variable x not in Γ, Γ' , p, q, s and t :*

1. *If $\Gamma, \Gamma' \models p \Rightarrow q$, then $\Gamma, x : r, \Gamma' \models p \Rightarrow q$.*
2. *If $\Gamma, \Gamma' \vdash s <: t$, then $\Gamma, x : r, \Gamma' \vdash s <: t$.*
3. *If $\Gamma, \Gamma' \vdash e : t$, then $\Gamma, x : r, \Gamma' \vdash e : t$.*

The first weakening lemma can be proved by observing that we can always construct a denotation θ' of Γ, Γ' from a denotation θ of $\Gamma, x : t, \Gamma'$, by removing the x entry from θ . Since x is fresh, removing it from θ does not impact the validity of this implication. With the first weakening lemma in hand, the remaining two are straightforward.

4.3.2 Progress & Preservation

► **Lemma 12** (Progress). *If $\emptyset \vdash e : t$ then e is a value or $\exists e'. e \rightsquigarrow e'$.*

The proof is done by induction on typing, following the standard approach of FJ.

► **Lemma 13** (Preservation). *If $\emptyset \vdash e : t$ and $e \rightsquigarrow e'$, then $\emptyset \vdash e' : t$.*

The proof is done by induction on the typing judgment and using the structural lemmas for substitutions and environment narrowings. To argue the preservation in the presence of general selfification and type substitution, Lemma 6 and Lemma 4 must also be utilized.

4.3.3 Closing Substitution

► **Lemma 14** (Closing Substitution). *If $\Gamma \vdash e : t$, then $\forall \theta \in [\Gamma]. \emptyset \vdash \theta(e) : \theta(t)$.*

The closing substitution lemma bears a similarity with the substitution lemma (Lemma 10). They both concern the invariance of typing under substitution. The closing substitution lemma can be proved by induction on typing. Most of the cases are standard, except for the variable case, which requires proving $\emptyset \vdash \theta(x) : \theta(t)$ under $\Gamma \vdash x : t$. Since θ is a denotation of Γ , we know that x must be in θ and $\theta(x) \in [\theta(t)]$. Thus, Lemma 8 can be applied to construct the expected typing judgment.

4.4 Type Soundness

To improve the readability, we reproduce Type Soundness (Theorem 1) below:

► **Corollary 15** (Type Soundness). *If $\emptyset \vdash e : t$ and $e \rightsquigarrow^* e'$, then e' is a value or $\exists e''. e' \rightsquigarrow e''$.*

Type soundness is an easy corollary of progress and preservation [70].

4.5 Logical Soundness

To improve readability, we reproduce the Logical Soundness (Theorem 2) below:

► **Corollary 16** (Logical Soundness). *If $\Gamma \vdash e : \{\nu : w | p\}$, $\vdash_w \Gamma$, and $\Gamma \downarrow e$, then $\Gamma \models [\nu \mapsto e]p$.*

The key to proving logical soundness is to observe that it can be reduced to closed logical soundness (shown below) if we can derive a corresponding closed typing judgment given any typing judgment. This is facilitated by the closing substitution lemma (Lemma 14).

► **Theorem 17** (Closed Logical Soundness). *If $\emptyset \vdash e : \{\nu : w | p\}$ and $\downarrow e$, then $\models [\nu \mapsto e]p$.*

Closed logical soundness is a natural consequence of preservation and typing denotation. Supposing e evaluates to v , the proof skeleton is that:

1. Due to the preservation lemma, $\emptyset \vdash v : \{\nu : w | p\}$.
2. Due to the typing denotation lemma, $v \in [\{\nu : w | p\}]$, thus $\models [\nu \mapsto v]p$.
3. Lastly, we can apply the evaluation invariant lemma to get $\models [\nu \mapsto e]p$.

5 Logical Encoding: LFJ

Following the standard procedure as outlined in, e.g., [6], we convert RFJ to an algorithmic bidirectional type system. The only judgment whose algorithmic property was unexplored was the class-based refinement subtyping. In this section, we present an encoding of RFJ to an order-sorted first-order logic [57], named LFJ, which gives a convenient axiomatic approach to determine RFJ refinement subtyping by invoking logical decision procedures.

5.1 Language

Figure 9 presents the syntax of LFJ. The constant symbols (c) are for RFJ values. We assume each RFJ value has a corresponding LFJ constant symbol. The function symbols (g) are for methods (N_m), field selectors (C_f), class constructors (C_{cr}), and primitive operations in RFJ. We associate methods with nominal names and field selectors with class names, for attributing more precise semantics (detailed later). Note that interfaces have no fields. The terms in LFJ do not contain quantification: they are viewed as implicitly quantified and a universal quantification would be added to the outermost to close them. Sorts in LFJ consist of \top , Int , $Bool$, and N . The sorts have an apparent correspondence with RFJ base types. We denote $|w|$ as the translation of a base type w to its sort, and $|t|$ as the translation from a refinement type t to its sort. The subsort relation \sqsubseteq is straightforwardly translated from the base-subtyping relation. The signatures of functions are also translated from their RFJ type definitions, e.g., the signature of C_m is $C \rightarrow |t| \rightarrow |r|$ if $mtype(m, C) = p \rightarrow x : t \rightarrow r$.

Syntax		Term Translation
$e, p ::=$	<i>terms:</i>	
x	<i>variable</i>	$ x = x$
c	<i>constant</i>	$ v = c_v$
$g(\bar{e})$	<i>apply</i>	$ \neg e_0 = \neg(e_0)$
$let x = e in e$	<i>let binding</i>	$ e_0 \oplus e_1 = \oplus(e_0 , e_1)$
$g ::= N_m \mid C_f \mid C_{cr} \mid \neg \mid \oplus$	<i>functions</i>	$ new C(\bar{e}) = C_{cr}(\bar{e})$
		$ e_0.m(e_1) = \delta(e_0)_m(e_0 , e_1)$
		$ e_0.f = \delta(e_0)_f(e_0)$
		$ let x = e_0 in e = let x = e_0 in e $
<i>Sorts</i>		
$s ::= \top \mid Int \mid Bool \mid N$	<i>sorts</i>	
$ w = match w with$	<i>base translation</i>	
$ \top \Rightarrow \top int \Rightarrow Int bool \Rightarrow Bool N \Rightarrow N$		$ \emptyset = true$
$ t = [t] $	<i>type translation</i>	$ \Gamma, x : \{\nu : u p\} = \Gamma \wedge [\nu \mapsto x]p $
$\sqsubseteq \doteq <:_b $	<i>subsort</i>	
		<i>Environment Translation</i>

Figure 9 LFJ syntax and translation.

Translation. The translation from RFJ terms and type environments to LFJ terms is mostly straightforward. The only thing to note is the association of type information during the translation of method invocations and field accesses, marked **brown** in Figure 9. This is facilitated by the *typeof* function: $\delta(e)$ is the static type of expression e . δ can be constructed during type checking. The association of type information is important for two purposes (we take method invocations as an example, but the argument also applies to field accesses):

- *Disambiguation.* Suppose the method m is defined by two classes C and D , which share no common superclass except `Object`. If methods are not associated with nominal types, the LFJ function representation of m would necessitate an assumed domain of *Object* for its first parameter, rendering the model for the function inherently partial, because not all *Object* has an m implementation. Incorporating type information ensures model totality for the first parameter by guaranteeing the existence of at least one implementation of m ; such existence is verified by static type checking. This totality guarantee plays an important role in the intended model (c.f., Section 5.2).
- *Axiomatization.* The aim of LFJ is to provide an axiomatization of its intended model (c.f., Section 5.3). By associating type information, the axiomatization can be crafted with greater specificity and accuracy.

5.2 Intended Model

Domain:	Functions:
$G_I = G_{\top} = \emptyset$	$\neg, \wedge, \vee = normal$
$G_C = \{C(\bar{d}_s) \mid \bar{d}_s \in \bar{D}_{ t }\}, fs(C) = \bar{t} \bar{f}$	$C_{cr}(\bar{d}) = C(\bar{d})$
$G_{Int} = \mathcal{Z}$	$C_{fi}(C'(\bar{d})) = d_i, C' \sqsubseteq C \text{ and } fs(C) = \bar{t} \bar{f}$
$G_{Bool} = \{T, F\}$	$N_m(this, x) = \begin{cases} \llbracket mb(m, C) \rrbracket(this, x) \text{ if } this = C(\bar{d}) \\ \dots \text{ proceeds for all } C \sqsubseteq N \end{cases}$
$D_s = \{d \mid d \in G_{s'} \wedge s' \sqsubseteq s\}$	

Figure 10 The intended model of LFJ. fs is short for *fields*.

In this section, we delineate the construction of an intended model \mathcal{A} for LFJ, given in Figure 10. This model bears similarities with several denotational semantics of class-based languages [58, 12], especially in the usage of *conditional functions* as models of method invocations, whereas we work with order-sorted logic, different from those semantics.

Domains. Each sort s is associated with a dynamic domain G_s and a static domain D_s . The dynamic domain of a sort is a *set* containing all values inherently belonging to the sort. The dynamic domains of \top and I (i.e., interfaces) are both \emptyset . G_{Int} and G_{Bool} are standard. G_C is the finite term trees [22] generated in a sort-correct manner (i.e., each field is drawn from the static domain of the corresponding sort). The static domain (or simply, domain) for a sort s aggregates the dynamic domains of its subsorts, as in standard OS-FOL [57].

Functions. The model adopts conventional interpretations for equality and boolean operators. The intended functions for constructors and fields are the constructing and destructing functions for term trees. The intended function of N_m is just a *conditional function* composed of the denotations of the implementation functions conditioned by the first parameter (i.e., the receiving object). We do not detail the denotations in this paper: because we require termination for well-typed RFJ programs, those denotations are total on their domains and can be constructed using standard fixed-point techniques as shown in, e.g, [46].

Algorithmic Subtyping. With the intended model \mathcal{A} in hand, we now define the algorithmic subtyping relation:

$$\frac{w <_b u \quad \mathcal{A} \models_L \forall \bar{x}. |\Gamma| \wedge |p| \Rightarrow |q|}{\Gamma \vdash \{\nu : w|p\} <_L \{\nu : u|q\}} A\text{-Subtyping}$$

where \models_L is the normal semantics of OS-FOL [57]. We assume all variables in Γ are distinct and are not ν . We use a universal quantification $\forall \bar{x}$ to close the formula, where \bar{x} is the variables used in Γ , p and q .

We establish the soundness of the algorithmic subtyping with respect to the refinement subtyping, which is a corollary of the semantic equivalence and translation-substitution distributivity. Semantic equivalence states the true sentences in RFJ logical interpretation are also true in \mathcal{A} , and vice versa. Translation-substitution distributivity states it does not matter whether we apply a closing substitution prior to or after the translation.

- **Proposition 18** (Semantic Equivalence). $\mathcal{A} \models_L |p| \Leftrightarrow \models p$
- **Proposition 19** (Translation-substitution Distributivity). $\mathcal{A} \models_L |\theta|(|p|) \Leftrightarrow \mathcal{A} \models_L |\theta(p)|$
- **Corollary 20.** If $\Gamma \vdash s <:_L t$, then $\Gamma \vdash s <: t$.

Proof. We give a brief proof sketch of Corollary 20 here. Suppose s is $\{\nu : w|p\}$ and t is $\{\nu : u|q\}$. To prove $\Gamma \vdash \{\nu : w|p\} <: \{\nu : u|q\}$, we need to prove $\forall \theta \in [\Gamma, \nu : w]. \text{if } \models \theta(p) \text{ then } \models \theta(q)$. By $\Gamma \vdash s <:_L t$, we have $\mathcal{A} \models_L \forall \bar{x}. |\Gamma| \wedge |p| \Rightarrow |q|$, which gives us $\forall \sigma. \mathcal{A} \models_L \sigma(|\Gamma| \wedge |p|) \Rightarrow \mathcal{A} \models_L \sigma(|q|)$ (by the semantics of OS-FOL). Pick σ as $|\theta|$, due to Propositions 18 and 19, we have $\mathcal{A} \models_L |\theta|(|\Gamma| \wedge |p|)$, which let us deduce $\mathcal{A} \models_L |\theta|(|q|)$. Using Propositions 18 and 19 again, but in the reverse direction, we have $\models \theta(q)$. ◀

5.3 Theory

To utilize the capability of deductive reasoning for checking subtyping algorithmically, we axiomatize the intended model \mathcal{A} by a theory $\mathcal{T}_{\mathcal{J}}$. $\mathcal{T}_{\mathcal{J}}$ includes the usual theory of Equality, Uninterpreted Functions, and Linear Integer Arithmetic (EUFLIA) [1]. Besides, it is equipped with axioms for N_m , C_f , and C_{cr} . We specify and explain them in this section.

- (1) *generate* : \rightarrow $\forall x : N. \bigvee_{C \sqsubseteq N} \exists \bar{y} : |fs(C)_t|. x = C(\bar{y})$
- (2) *inject* : \rightarrow $\forall \bar{x} : |fs(C)_t|, \bar{y} : |\bar{t}|. C_{cr}(\bar{x}) = C_{cr}(\bar{y}) \Rightarrow \bar{x} = \bar{y}$
- (3) *discriminate* : $C \neq D \rightarrow \forall \bar{x} : |fs(C)_t|, \bar{y} : |fs(D)_t|. C_{cr}(\bar{x}) \neq D_{cr}(\bar{y})$
- (4) *access* : $fs(C) = \bar{f} \bar{t}, C' \sqsubseteq C \rightarrow \forall \bar{x} : |fs(C')_t|. C_{fi}(C'_{cr}(\bar{x})) = x_i$
- (5) *invoke* : $mt(m, N)_x = t_x, C \sqsubseteq N, mb(m, C) = (x, e) \rightarrow$
 $\forall o : N, x : |t_x|, \bar{d} : |fs(C)_t|. o = C(\bar{d}) \Rightarrow N_m(o, x) = |e|$

The above listing gives five axiom schemata. The symbol \rightarrow means “instantiate”: if the condition on the left is satisfied, one can instantiate an axiom following the schema on the right. The symbols fs , mb , fs_t , and mt_x short for *fields*, *mbody*, the type part of *fields*, and the parameter part of *mtype* (or *mtypei* for interfaces), respectively.

The axiom schemata are straightforward given the intended model \mathcal{A} . However, they may not be as efficient as we want. To address this, we add two derivable properties as axioms, to speed up deductive reasoning. The first covers cases where the branches of a method direct to the same implementation. We have seen such a case in our example: *Anchovy.remA* has two branches that direct to the same implementation. We call these methods like *Anchovy.remA final methods*. Final methods have the same implementation on all branches, and there is no need to actually do the branching. We axiomatize their semantics using the axiom schema (6) shown below. An instantiation of (6) gives the property p_3 we discussed in Section 2.2. The second covers cases where a method is called on a subclass of the declared type. For

example, suppose we have $\nu = Pi_{remA}(x)$ and $x : An$, and we want to prove $\nu = An_{remA}(x)$. With basic axioms (1) through (5) above, we have to first deduce the fact that x can only be $An(\dots)$ or $Ma(\dots)$, then analyze the semantics of Pi_{remA} and An_{remA} for those two cases, and finally deduce that the equality holds for both cases. However, this is mostly redundant: Pi_{remA} and An_{remA} are the same function if the first argument is known to be an An . We axiomatize this fact using schema (7) shown below. For certain cases involving comparing method-call results, axiom schema (7) can speed up reasoning significantly.

- (6) $final : mt(m, C)_x = t_x, C.m \text{ final}, mb(m, C) = (x, e), \Rightarrow \forall o : C, x : |t_x|. C_m(o, x) = |e|$
- (7) $override : N' \sqsubseteq N, mt(m, N')_x = t_x \Rightarrow \forall o : N', x : |t_x|. N_m(o, x) = N'_m(o, x)$

Encoding into Many-sorted Logic. The aforementioned axioms are defined in OS-FOL and should be used in order-sorted deductive reasoning. Unfortunately, we are not aware of any SMT solver that supports order-sortedness. Thus, we translate the axioms into many-sorted logic following the strategy suggested by Leino [38]. The translation of primitive data types is straightforward. For objects, a unified sort *Object* is designated. We then introduce a sort *Nominal* to encompass all nominal entities, i.e., classes and interfaces in the targeting RFJ program. We also declare the sub-nominal relation between those entities. The association of nominal information with objects is facilitated through the *Tag* function, which relates objects with their nominal identifiers. The sort requirements become sub-nominal checkings on tags, e.g., instead of $\forall x : C. p(x)$, we use $\forall x : Object. sub-nominal(Tag(x), C) \Rightarrow p(x)$.

The direct encoding of the sort \top into many-sorted logic is beyond our current scope, primarily influencing the polymorphic nature of equality. Nevertheless, given the uniform *Object* sort for all object values, object equality is still \top -typed essentially, circumventing potential limitations posed by the absence of a direct \top sort.

6 Mechanization and Implementation

6.1 Coq Mechanization

We mechanize the meta-theory of RFJ in Coq. There are two major technical challenges around the mechanization. (1) *Binders*. Handling binders is cumbersome and complex [3], especially considering the number of binder structures present in RFJ (e.g., methods, let-bindings, and refinement types). To address this issue, we adopt the locally nameless representation [13]. Although the locally nameless representation has been widely used in mechanizing functional languages [8, 29, 13], to the best of our knowledge, ours is the first mechanization of a class-based language that utilizes this technique. (2) *Nested Inductive Types*. The presence of nested inductive types within our definitions poses a significant challenge; that is, the default induction principles generated by Coq fell short when proving the most critical properties. To mitigate this issue, we specify the custom induction principles for a range of inductive definitions (e.g., terms, typing judgments, and big-step semantics), following the classical methodology [15].

We briefly overview the structure of the mechanization, which contains about 15K lines of Coq code:

1. Definitions (3K): language definitions as presented in Section 3.
2. Lemmas (11K):
 - a. Basic Lemmas (5K): miscellaneous lemmas concerning basic operations, semantics, and class/interface definitions (some of which are listed in Section 4.1).
 - b. Logical Lemmas (2K): lemmas concerning the logical interpretation (c.f., Section 4.2).
 - c. Typing Lemmas (4K): basic, structural, and crucial lemmas of typing (c.f., Section 4.3).
3. Theorems (1K): type and logical soundness theorems (c.f., Sections 4.4 and 4.5).

6.2 Python Implementation

We implement a refinement type checker for RFJ. The implementation is written in roughly 2,000 lines of Python code, with Z3 [19] as the SMT backend. In addition to all features of RFJ, the type checker also supports a form of *if-then-else* following the standard practice [32], to increase the scope of the evaluation. The concrete syntax supported in the implementation is a subset of Python with static types. We opt for Python just to reuse its parser and editor supports. RFJ can be implemented for any other class-based language.

To test the type checker, we handcraft a test suite, including all the major examples that do not use type-test/downcast or imperative features from a Java textbook [23], as well as some interesting examples inspired by previous work [65]. Each example is paired with some non-trivial properties. In total, there are 14 examples with about 1,500 LOC, covering all important features of RFJ. We list several representative examples in Table 1.

Table 1 Several representative examples.

Name	Features	LOC	Properties
pizza	classes, overrides	135	remA_noinc_price, remA_idempotent
pizza visitor	visitors, upcasts	110	noObj_after_rem, noObj_after_effective_sub
tree	visitor interfaces	152	height_ge_root
geometry	factory methods	184	origin_in_shape
list	data structures	125	contains_weakening, inserts_preserve_sortedness
λ calculus	data structures	71	size_positive, substitution_nodec_size
stlc	meta-theories	307	map_extend_included, typing_weakening

Type-checking each example took under 5 seconds, on an Apple M1 machine.

7 Discussion

In this section, we discuss specific designs of RFJ in greater detail.

Type Substitution vs ANF and Existential types. In the realm of refinement type systems, the conventional strategy often involves leveraging ANF [32, 37] or existential types [47, 34, 8] to maintain the logic of refinements within a decidable framework, such as EUFLIA [9]. Our approach, however, consciously eschews these mechanisms and sticks to simple type substitution for three compelling reasons. (1) *From the theoretical perspective.* We want to argue the soundness of our system within a broader, more generalized framework: all RFJ programs expressed in ANF are inherently valid within our system, while the converse does not hold. Thus, our results perfectly apply to the condition where ANF is required (e.g., a particular implementation may perform ANF transformation before type checking). (2) *From the algorithmic perspective.* Recent advances [41, 44] have shown a complete algorithm for formula validity under a user-specified theory exists, which is exactly what we need to perform algorithmic subtyping checking. The fact that all our examples are checked costing only a little time also evidences that a reasonably efficient algorithm exists even if the logic falls outside the familiar decidable fragment. (3) *From the pragmatical perspective.* Eliminating ANF and existential types significantly lowers the barrier between the programmer’s intent and the underlying type system, simplifying the debugging process. To further lower the barrier, our typing rules are carefully formulated without using any implicit environment extension (e.g., the *Field* and *Invoke* rules in [47]). The only cases that would extend the typing and subtyping environment are *Let* and method typing, thereby maintaining a clear correspondence between the code and its type-level representation.

Axiomatization vs. Reflection. As pointed out by prior work [66], there are two kinds of methodologies to support user-defined functions in refinement type systems: axiomatization and reflection. Axiomatization articulates the semantics of user-defined functions through logical axioms, an approach we adopt and have elaborated on in Section 5.3. In contrast, reflection directly incorporates the function definition into the return type’s refinement (e.g., the return type of `Anchovy.remA` can be declared as $\{\nu : \text{Pizza} | \nu = \text{this}.p.\text{remA}()\}$ to reflect its definition). In our system, programmers can utilize reflection by manually specifying the method return type (reflection annotation could also be provided to automate this process). Those reflections are always valid thanks to general reflection, which ensures that terms are always recorded in refinements. Notably, reflection offers an alternative to the *final* constraint of the invoke axiom schema (c.f. Section 5.3): one can reflect the definition of an overriding method and the overridden method simultaneously, as long as the return types of those methods obey the co-variance principle.

The major difference between reflection and axiomatization resides in the instantiation strategy of method definitions. With reflection, instantiations of the reflected functions are performed within the type system, either by the programmer or an algorithm (e.g., PLE in [66]). With axiomatization, instantiation is delegated to the SMT solver, although special mechanisms such as *trigger/fuel* [2, 40] are needed to keep the process in control. Currently, no special algorithm or mechanism for reflection or axiomatization is employed in RFJ. However, we identify the comparison of these two methodologies in RFJ, especially in the context of a reflection instantiation algorithm and more advanced type system features (e.g., occurrence typing and union/intersection types) as important future work.

8 Related Work

This work intersects three research topics: class-based refinement type systems, mechanization of refinement types and class-based languages, and SMT-based reasoning in program verifiers.

Class-based Refinement Type Systems. Class is an important and time-honored abstraction in object-oriented programming [16, 59, 25, 31], with numerous pieces of literature devoted [69, 60, 5, 55] to its extensions. In particular, many works have focused on class-based refinement type systems. For example, Nystrom et al. [47] formalize core X10 as a refinement type system. However, they focus only on the functional aspects. Vekris et al. [67] introduce a refinement type calculus that not only conducts immutability analysis but also integrates union and intersection types, with the caveat that only immutable fields are subject to refinement. Campos et al. [10] combine refinement types with class-based linear types, further increasing the support for imperative features. Kuncak et al. [56] present qualified type, a form of refinement type, and offer an in-depth discussion on qualifier inference. Gamboa et al. [26] address the practical challenges of incorporating refinement types into existing class-based systems by proposing a design approach to usability.

All the aforementioned work limits their refinements to well-established decidable SMT theories (e.g., EUFLIA), and thus have significant issues concerning soundness and expressiveness, as we have explained before. Meanwhile, although there are systems [33, 63] exploring the support for more expressive refinements, their approach is mainly pragmatic (i.e., they both rely on external verification tools to support the expressive refinements), which complicates the analysis of their meta-theoretical properties further.

This work addresses the expressiveness and soundness issues in a fundamental way, by providing an expressive and mechanized calculus grounded in Featherweight Java. We anticipate that extensions such as generics and imperative features could be seamlessly integrated into our framework, prospects we reserve for future exploration.

Mechanization of Refinement Types and Class-based Languages. Several pieces of recent work have been dedicated to the mechanization of refinement types. Lehmann et al. [37] formalize a refinement type system in Coq. Their logical interpretation is axiomatized via a few basic requirements. This interpretation, however, leaves the semantics of logical formulas nebulous. Meanwhile, their proof focuses solely on the closed logical soundness, rather than general logical soundness. Wang et al. [68] mechanize in Coq a calculus that uses refinement types for complexity analysis, defining logical interpretations through denotational semantics that link refinements to Coq definitions. This method restricts the scope of terms that can be utilized as refinements due to the limitation of denotational semantics. Borkowski et al. [8] mechanizes a polymorphic refinement type system in Coq. They use an axiomatized logical interpretation for type soundness, and an operational-semantics-based logical interpretation for logical soundness. Hamza et al. [29] formalize a polymorphic refinement type system in Coq. They also employ an operational-semantics-based logical interpretation (named reducibility in the original paper). Our work draws inspiration from the two works on using operational-semantics-based logical interpretations, yet our proof diverges notably, especially given the inapplicability of logical relation techniques in our context. Moreover, our framework includes several special mechanisms such as general selfification and nominal subtyping, extending beyond the capabilities of the systems devised by those authors. Chen’s work [14] in Agda takes a unique route by integrating Agda to define a denotational semantics for refinements. However, the algorithmic properties are complicated, due to the reliance on Agda’s logic. Ghalayini et al. [26] opt for a categorical-theoretical perspective for logical interpretation in their mechanized refinement type system in Lean [18], contrasting with the semantic logical interpretation in our work.

Apart from the abovementioned differences, our research sets itself apart by focusing on a class-based calculus. This foundation renders our model particularly adept at mirroring object-oriented programming paradigms, a facet not directly addressed by the aforementioned mechanizations. There are also several mechanizations of class-based languages [42, 20, 17]. However, neither of them supports refinement types.

SMT-based Deductive Reasoning in Program Verifiers. Since its inception, SMT solvers have played a pivotal role in the automated verification of **functional** properties. Simplify [21] and ESC/Java [24] are among the earliest examples. Subsequently, a wave of advanced program verifiers like Dafny [39], Leon/Stainless [7, 29], F* [61] and Liquid Haskell [65, 66] have garnered attention in both academia and industry. Among those systems, Dafny and Leon/Stainless all support some object-oriented constructs. However, they lack RFJ’s support for nominal subtyping and method inheritance. Recent scholarly work has delved into the foundational aspects of SMT-based deductive reasoning, focusing especially on the completeness problem [44, 41, 45]. However, the arguments of those papers are all set upon many-sorted logic, diverging from the order-sorted logic in our study.

On the other hand, SMT solvers have also been extensively used in verifying **heap** properties. The modeling and verification of those properties (typically in separation logic [48, 49]) are, in general, beyond the ability of vanilla SMT theories [43]. Despite these challenges, research has successfully identified certain significant fragments yielding effective decision procedures falling into the SMT realm [43, 54, 53]. Currently, RFJ is a purely functional calculus. However, we believe that it is promising to incorporate those advancements to support the reasoning of heaps, considering imperative features are ambitious in class-based object-oriented languages [50].

9 Conclusion and Future Work

This paper introduces Refinement Featherweight Java (RFJ), advancing class-based refinement types with expressive refinements for comprehensive logical constraints. We mechanize RFJ in Coq, proving its soundness rigorously. We bridge the declarative calculus and algorithmic verification via a specified fragment in OS-FOL, making RFJ’s refinements accessible for SMT reasoning. The deliberate choice of FJ and OS-FOL for our fundamental framework facilitates important future extensions, such as polymorphic and imperative features, and a thorough exploration of algorithmic properties.

References

- 1 Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive software verification – the KeY book – from theory to practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016. doi:[10.1007/978-3-319-49812-6](https://doi.org/10.1007/978-3-319-49812-6).
- 2 Nada Amin, K. Rustan M. Leino, and Tiark Rompf. Computing with an SMT solver. In Martina Seidl and Nikolai Tillmann, editors, *Tests and Proofs – 8th International Conference, TAP@STAF 2014, York, UK, July 24–25, 2014. Proceedings*, volume 8570 of *Lecture Notes in Computer Science*, pages 20–35. Springer, 2014. doi:[10.1007/978-3-319-09099-3_2](https://doi.org/10.1007/978-3-319-09099-3_2).
- 3 Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The poplmark challenge. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22–25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2005. doi:[10.1007/11541868_4](https://doi.org/10.1007/11541868_4).
- 4 Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification – 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14–20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011. doi:[10.1007/978-3-642-22110-1_14](https://doi.org/10.1007/978-3-642-22110-1_14).
- 5 Lorenzo Bettini, Viviana Bono, Mariangiola Dezani-Ciancaglini, Paola Giannini, and Betti Venneri. Java & lambda: a featherweight story. *Log. Methods Comput. Sci.*, 14(3), 2018. doi:[10.23638/LMCS-14\(3:17\)2018](https://doi.org/10.23638/LMCS-14(3:17)2018).
- 6 Gavin M. Bierman, Andrew D. Gordon, Catalin Hritcu, and David E. Langworthy. Semantic subtyping with an SMT solver. *J. Funct. Program.*, 22(1):31–105, 2012. doi:[10.1017/S0956796812000032](https://doi.org/10.1017/S0956796812000032).
- 7 Régis Blanc, Viktor Kuncak, Etienne Kneuss, and Philippe Suter. An overview of the leon verification system: verification by translation to recursive functions. In *Proceedings of the 4th Workshop on Scala, SCALA@ECOOP 2013, Montpellier, France, July 2, 2013*, pages 1:1–1:10. ACM, 2013. doi:[10.1145/2489837.2489838](https://doi.org/10.1145/2489837.2489838).
- 8 Michael Borkowski, Niki Vazou, and Ranjit Jhala. Mechanizing refinement types. *Proc. ACM Program. Lang.*, 8(POPL):2099–2128, 2024. doi:[10.1145/3632912](https://doi.org/10.1145/3632912).
- 9 Aaron R. Bradley and Zohar Manna. *The calculus of computation – decision procedures with applications to verification*. Springer, 2007. doi:[10.1007/978-3-540-74113-8](https://doi.org/10.1007/978-3-540-74113-8).
- 10 Joana Campos and Vasco T. Vasconcelos. Dependent types for class-based mutable objects. 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 13:1–13:28. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:[10.4230/LIPICS.ECOOP.2018.13](https://doi.org/10.4230/LIPICS.ECOOP.2018.13).
- 11 Giuseppe Castagna. Covariance and contravariance: Conflict without a cause. *ACM Trans. Program. Lang. Syst.*, 17(3):431–447, 1995. doi:[10.1145/203095.203096](https://doi.org/10.1145/203095.203096).

- 12 Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A semantics for lambda-&-early: A calculus with overloading and early binding. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings*, volume 664 of *Lecture Notes in Computer Science*, pages 107–123. Springer, 1993. doi: [10.1007/BF0037101](https://doi.org/10.1007/BF0037101).
- 13 Arthur Charguéraud. The locally nameless representation. *J. Autom. Reason.*, 49(3):363–408, 2012. doi: [10.1007/S10817-011-9225-2](https://doi.org/10.1007/S10817-011-9225-2).
- 14 Zilin Chen. A hoare logic style refinement types formalisation. In *TyDe '22: 7th ACM SIGPLAN International Workshop on Type-Driven Development, Ljubljana, Slovenia, 11 September 2022*, pages 1–14. ACM, 2022. doi: [10.1145/3546196.3550162](https://doi.org/10.1145/3546196.3550162).
- 15 Adam Chlipala. *Certified programming with dependent types – a pragmatic introduction to the Coq Proof Assistant*. MIT Press, 2013. URL: <http://mitpress.mit.edu/books/certified-programming-dependent-types>.
- 16 William R. Cook. On understanding data abstraction, revisited. 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 557–572. ACM, 2009. doi: [10.1145/1640089.1640133](https://doi.org/10.1145/1640089.1640133).
- 17 Samuel da Silva Feitosa, Rodrigo Geraldo Ribeiro, and André Rauber Du Bois. Towards an extrinsic formalization of featherweight java in agda. *CLEI Electron. J.*, 24(3), 2021. doi: [10.19153/CLEIEJ.24.3.3](https://doi.org/10.19153/CLEIEJ.24.3.3).
- 18 Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28 – 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021. doi: [10.1007/978-3-030-79876-5_37](https://doi.org/10.1007/978-3-030-79876-5_37).
- 19 Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi: [10.1007/978-3-540-78800-3_24](https://doi.org/10.1007/978-3-540-78800-3_24).
- 20 Benjamin Delaware, William R. Cook, and Don S. Batory. Product lines of theorems. 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 595–608. ACM, 2011. doi: [10.1145/2048066.2048113](https://doi.org/10.1145/2048066.2048113).
- 21 David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005. doi: [10.1145/1066100.1066102](https://doi.org/10.1145/1066100.1066102).
- 22 Khalil Djelloul, Thi-Bich-Hanh Dao, and Thom W. Frühwirth. Theory of finite or infinite trees revisited. *Theory Pract. Log. Program.*, 8(4):431–489, 2008. doi: [10.1017/S1471068407003171](https://doi.org/10.1017/S1471068407003171).
- 23 Matthias Felleisen and Daniel P. Friedman. *A little Java, a few patterns*. MIT Press, 1996.
- 24 Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In Jens Knoop and Laurie J. Hendren, editors, *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, pages 234–245. ACM, 2002. doi: [10.1145/512529.512558](https://doi.org/10.1145/512529.512558).
- 25 Maurizio Gabbielli, Simone Martini, and Saverio Giallorenzo. *Programming languages: principles and paradigms, Second Edition*. Undergraduate Topics in Computer Science. Springer, 2023. doi: [10.1007/978-3-031-34144-1](https://doi.org/10.1007/978-3-031-34144-1).
- 26 Catarina Gamboa, Paulo Canelas, Christopher Steven Timperley, and Alcides Fonseca. Usability-oriented design of liquid types for java. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 1520–1532. IEEE, 2023. doi: [10.1109/ICSE48619.2023.00132](https://doi.org/10.1109/ICSE48619.2023.00132).

- 27 Simon J. Gay, Nils Gesbert, António Ravara, and Vasco Thudichum Vasconcelos. Modular session types for objects. *Log. Methods Comput. Sci.*, 11(4), 2015. doi:10.2168/LMCS-11(4:12)2015.
- 28 James Gosling, William N. Joy, and Guy L. Steele Jr. *The Java Language Specification*. Addison-Wesley, 1996.
- 29 Jad Hamza, Nicolas Voirol, and Viktor Kuncak. System FR: formalized foundations for the stainless verifier. *Proc. ACM Program. Lang.*, 3(OOPSLA):166:1–166:30, 2019. doi:10.1145/3360592.
- 30 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.
- 31 Bart Jacobs. Objects and classes, co-algebraically. In Burkhard Freitag, Cliff B. Jones, Christian Lengauer, and Hans-Jörg Schek, editors, *Object Orientation with Parallelism and Persistence (the book grow out of a Dagstuhl Seminar in April 1995)*, pages 83–103. Kluwer Academic Publishers, 1995.
- 32 Ranjit Jhala and Niki Vazou. Refinement types: A tutorial. *Found. Trends Program. Lang.*, 6(3-4):159–317, 2021. doi:10.1561/2500000032.
- 33 Milod Kazerounian, Niki Vazou, Austin Bourgerie, Jeffrey S. Foster, and Emina Torlak. Refinement types for ruby. In Isil Dillig and Jens Palsberg, editors, *Verification, Model Checking, and Abstract Interpretation – 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings*, volume 10747 of *Lecture Notes in Computer Science*, pages 269–290. Springer, 2018. doi:10.1007/978-3-319-73721-8_13.
- 34 Kenneth L. Knowles and Cormac Flanagan. Compositional reasoning and decidable checking for dependent contract types. In Thorsten Altenkirch and Todd D. Millstein, editors, *Proceedings of the 3rd ACM Workshop Programming Languages meets Program Verification, PLPV 2009, Savannah, GA, USA, January 20, 2009*, pages 27–38. ACM, 2009. doi:10.1145/1481848.1481853.
- 35 Daniel Kroening and Ofer Strichman. *Decision procedures – an algorithmic point of view*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008. doi:10.1007/978-3-540-74105-3.
- 36 Florian Lanzinger, Alexander Weigl, Matthias Ulbrich, and Werner Dietl. Scalability and precision by combining expressive type systems and deductive verification. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–29, 2021. doi:10.1145/3485520.
- 37 Nico Lehmann and Éric Tanter. Formalizing simple refinement types in coq. In *2nd International Workshop on Coq for Programming Languages (CoqPL’16), St. Petersburg, FL, USA, 2016*.
- 38 K Rustan M Leino. This is boogie 2. *manuscript KRML*, 178(131):9, 2008.
- 39 K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning – 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010. doi:10.1007/978-3-642-17511-4_20.
- 40 K. Rustan M. Leino and Clément Pit-Claudel. Trigger selection strategies to stabilize program verifiers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification – 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, volume 9779 of *Lecture Notes in Computer Science*, pages 361–381. Springer, 2016. doi:10.1007/978-3-319-41528-4_20.
- 41 Christof Löding, P. Madhusudan, and Lucas Peña. Foundations for natural proofs and quantifier instantiation. *Proc. ACM Program. Lang.*, 2(POPL):10:1–10:30, 2018. doi:10.1145/3158098.
- 42 Julian Mackay, Hannes Mehnert, Alex Potanin, Lindsay Groves, and Nicholas Cameron. Encoding featherweight java with assignment and immutability using the coq proof assistant. In Wei-Ngan Chin and Aquinas Hobor, editors, *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs, FTfJP 2012, Beijing, China, June 12, 2012*, pages 11–19. ACM, 2012. doi:10.1145/2318202.2318206.

- 43 P. Madhusudan, Gennaro Parlato, and Xiaokang Qiu. Decidable logics combining heap structures and data. 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 611–622. ACM, 2011. doi:10.1145/1926385.1926455.
- 44 Adithya Murali, Lucas Peña, Ranjit Jhala, and P. Madhusudan. Complete first-order reasoning for properties of functional programs. *Proc. ACM Program. Lang.*, 7(OOPSLA2):1063–1092, 2023. doi:10.1145/3622835.
- 45 Adithya Murali, Lucas Peña, Christof Löding, and P. Madhusudan. A first-order logic with frames. *ACM Trans. Program. Lang. Syst.*, 45(2):7:1–7:44, 2023. doi:10.1145/3583057.
- 46 Hanne Riis Nielson and Flemming Nielson. *Semantics with applications*, volume 104. Springer, 1992.
- 47 Nathaniel Nystrom, Vijay A. Saraswat, Jens Palsberg, and Christian Grothoff. Constrained types for object-oriented languages. 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 457–474. ACM, 2008. doi:10.1145/1449764.1449800.
- 48 Peter W. O’Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007. doi:10.1016/J.TCS.2006.12.035.
- 49 Peter W. O’Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg, editor, *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001. doi:10.1007/3-540-44802-0_1.
- 50 Johan Östlund and Tobias Wrigstad. Welterweight java. In Jan Vitek, editor, *Objects, Models, Components, Patterns, 48th International Conference, TOOLS 2010, Málaga, Spain, June 28 - July 2, 2010. Proceedings*, volume 6141 of *Lecture Notes in Computer Science*, pages 97–116. Springer, 2010. doi:10.1007/978-3-642-13953-6_6.
- 51 Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In Jean-Jacques Lévy, Ernst W. Mayr, and John C. Mitchell, editors, *Exploring New Frontiers of Theoretical Informatics, IFIP 18th World Computer Congress, TC1 3rd International Conference on Theoretical Computer Science (TCS2004), 22-27 August 2004, Toulouse, France*, volume 155 of *IFIP*, pages 437–450. Kluwer/Springer, 2004. doi:10.1007/1-4020-8141-3_34.
- 52 Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.
- 53 Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating separation logic with trees and data. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification – 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 711–728. Springer, 2014. doi:10.1007/978-3-319-08867-9_47.
- 54 Andrew Reynolds, Radu Iosif, Cristina Serban, and Tim King. A decision procedure for separation logic in SMT. In Cyrille Artho, Axel Legay, and Doron Peled, editors, *Automated Technology for Verification and Analysis – 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*, volume 9938 of *Lecture Notes in Computer Science*, pages 244–261, 2016. doi:10.1007/978-3-319-46520-3_16.
- 55 Reuben N. S. Rowe and Steffen van Bakel. Semantic types and approximation for featherweight java. *Theor. Comput. Sci.*, 517:34–74, 2014. doi:10.1016/J.TCS.2013.08.017.
- 56 Georg Stefan Schmid and Viktor Kuncak. Smt-based checking of predicate-qualified types for scala. In Aggelos Biboudis, Manohar Jonnalagedda, Sandro Stucki, and Vlad Ureche, editors, *Proceedings of the 7th ACM SIGPLAN Symposium on Scala, SCALA@SPLASH 2016, Amsterdam, Netherlands, October 30 - November 4, 2016*, pages 31–40. ACM, 2016. doi:10.1145/2998392.2998398.

- 57 Peter H. Schmitt and Mattias Ulbrich. Axiomatization of typed first-order logic. In Nikolaj S. Bjørner and Frank S. de Boer, editors, *FM 2015: Formal Methods – 20th International Symposium, Oslo, Norway, June 24–26, 2015, Proceedings*, volume 9109 of *Lecture Notes in Computer Science*, pages 470–486. Springer, 2015. doi:10.1007/978-3-319-19249-9_29.
- 58 Thomas Studer. Constructive foundations for featherweight java. In Reinhard Kahle, Peter Schroeder-Heister, and Robert F. Stärk, editors, *Proof Theory in Computer Science, International Seminar, PTCS 2001, Dagstuhl Castle, Germany, October 7–12, 2001, Proceedings*, volume 2183 of *Lecture Notes in Computer Science*, pages 202–238. Springer, 2001. doi:10.1007/3-540-45504-3_13.
- 59 Ke Sun, Sheng Chen, Meng Wang, and Dan Hao. What types are needed for typing dynamic objects? A python-based empirical study. In Chung-Kil Hur, editor, *Programming Languages and Systems – 21st Asian Symposium, APLAS 2023, Taipei, Taiwan, November 26–29, 2023, Proceedings*, volume 14405 of *Lecture Notes in Computer Science*, pages 24–45. Springer, 2023. doi:10.1007/978-99-8311-7_2.
- 60 Ke Sun, Yifan Zhao, Dan Hao, and Lu Zhang. Static type recommendation for python. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10–14, 2022*, pages 98:1–98:13. ACM, 2022. doi:10.1145/3551349.3561150.
- 61 Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. Dependent types and multi-monadic effects in F. In Rastislav Bodík and Rupak Majumdar, editors, *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 256–270. ACM, 2016. doi:10.1145/2837614.2837655.
- 62 William W. Tait. Intensional interpretations of functionals of finite type I. *J. Symb. Log.*, 32(2):198–212, 1967. doi:10.2307/2271658.
- 63 Emina Torlak and Rastislav Bodík. Growing solver-aided languages with rosette. In Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld, editors, *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH ’13, Indianapolis, IN, USA, October 26–31, 2013*, pages 135–152. ACM, 2013. doi:10.1145/2509578.2509586.
- 64 Steffen van Bakel and Maribel Fernández. Normalization, approximation, and semantics for combinator systems. *Theor. Comput. Sci.*, 290(1):975–1019, 2003. doi:10.1016/S0304-3975(02)00548-0.
- 65 Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. Refinement types for haskell. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1–3, 2014*, pages 269–282. ACM, 2014. doi:10.1145/2628136.2628161.
- 66 Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. Refinement reflection: complete verification with SMT. *Proc. ACM Program. Lang.*, 2(POPL):53:1–53:31, 2018. doi:10.1145/3158141.
- 67 Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. Refinement types for typescript. In Chandra Krintz and Emery D. Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13–17, 2016*, pages 310–325. ACM, 2016. doi:10.1145/2908080.2908110.
- 68 Peng Wang, Di Wang, and Adam Chlipala. Timl: a functional language for practical complexity analysis with invariants. *Proc. ACM Program. Lang.*, 1(OOPSLA):79:1–79:26, 2017. doi:10.1145/3133903.
- 69 Stefan Wehr, Ralf Lämmel, and Peter Thiemann. Javagi : Generalized interfaces for java. In Erik Ernst, editor, *ECOOP 2007 – Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 – August 3, 2007, Proceedings*, volume 4609 of *Lecture Notes in Computer Science*, pages 347–372. Springer, 2007. doi:10.1007/978-3-540-73589-2_17.
- 70 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.

Information Flow Control in Cyclic Process Networks

Bas van den Heuvel 

HKA Karlsruhe, Germany

University of Freiburg, Germany

University of Groningen, The Netherlands

Farzaneh Derakhshan 

Illinois Institutie of Technology, Chicago, IL, USA

Stephanie Balzer 

Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

Protection of confidential data is an important security consideration of today's applications. Of particular concern is to guard against unintentional leakage to a (malicious) observer, who may interact with the program and draw inference from made observations. Information flow control (IFC) type systems address this concern by statically ruling out such leakage. This paper contributes an IFC type system for message-passing concurrent programs, the computational model of choice for many of today's applications such as cloud computing and IoT applications. Such applications typically either implicitly or explicitly codify protocols according to which message exchange must happen, and to statically ensure protocol safety, behavioral type systems such as session types can be used. This paper marries IFC with session typing and contributes over prior work in the following regards: (1) support of realistic cyclic process networks as opposed to the restriction to tree-shaped networks, (2) more permissive, yet entirely secure, IFC control, exploiting cyclic process networks, and (3) considering deadlocks as another form of side channel, and asserting deadlock-sensitive noninterference (DSNI) for well-typed programs. To prove DSNI, the paper develops a novel logical relation that accounts for cyclic process networks. The logical relation is rooted in linear logic, but drops the tree-topology restriction imposed by prior work.

2012 ACM Subject Classification Theory of computation → Linear logic; Security and privacy → Logic and verification; Theory of computation → Process calculi; Theory of computation → Type theory

Keywords and phrases Cyclic process networks, linear session types, logical relations, deadlock-sensitive noninterference

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.40

Related Version *Extended Version:* <https://arxiv.org/abs/2407.02304> [27]

Funding *Bas van den Heuvel:* Supported in part by the Dutch Research Council (NWO) under project No. 016.Vidi.189.046 (Unifying Correctness for Communicating Software).

Stephanie Balzer: Supported in part by the Air Force Office of Scientific Research under award number FA9550-21-1-0385 (Tristan Nguyen, program manager). Any opinions, findings and conclusions or recommendations expressed here are those of the author(s) and do not necessarily reflect the views of the U.S. Department of Defense.

1 Introduction

Many of today's emerging applications and systems such as cloud computing and IoT applications are inherently *concurrent* and *message passing*. Message passing also enjoys popularity in mainstream languages such as Erlang, Go, and Rust. Similar to functional languages with the λ -calculus as their theoretical model, the model of message-passing

 © Bas van den Heuvel, Farzaneh Derakhshan, and Stephanie Balzer;
licensed under Creative Commons License CC-BY 4.0

38th European Conference on Object-Oriented Programming (ECOOP 2024).

Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 40; pp. 40:1–40:30

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

concurrent languages is the process calculus [31, 47, 48]. A program in this setting amounts to a number of *processes* connected by *channels*, which compute by exchanging messages along these channels, rather than by β -reductions or writing to and reading from shared memory. Messages may even include channels themselves, a feature supported in the π -calculus [49, 59] and referred to as *higher-order* message passing.

Originally untyped [49], the π -calculus has gradually been enriched with types to prescribe the kinds of messages that can be exchanged over a channel [59] and to assert correctness properties, such as deadlock freedom and data-race freedom [40, 38, 39, 42]. Following in these footsteps, *session types* [32, 33] were conceived to additionally express the *protocols* underlying the exchange. Session types rely on a *linear* treatment of channels to model the state transitions induced by a protocol, which was even substantiated by a Curry-Howard correspondence between the session-typed π -calculus and linear logic [9, 66, 67, 64, 45, 10, 46]. Session types based on linear logic enjoy strong properties, comprising not only race and deadlock freedom but also protocol fidelity.

Security is another correctness consideration arising from today’s applications and systems. One security concern in particular is the protection of confidential information, by preventing unintentional leakage to a (malicious) observer, who may interact with the program and draw inference from made observations. Type systems for *information flow control (IFC)* rule out such leakage by type checking [65, 60, 57], given a lattice over security levels and the labeling of observables (e.g., output, locations, channels) with these levels. Well-typed programs then prevent “flows from high to low” and guarantee *noninterference*, i.e., that an observer cannot infer any secrets from made observations. To guarantee noninterference, advanced proof methods such as logical relations [54, 2, 22, 50, 37, 63] and bisimulations [44, 62, 61, 58] are used. If side channels [57], such as the termination channel, are present, then the literature distinguishes *progress-sensitive* noninterference (PSNI) from *progress-insensitive* noninterference (PINI), where the former only equates a divergent program run with another diverging one, whereas the latter equates a divergent program run with any other run [24].

Whereas the development of IFC type systems has been an active research field for decades for imperative and predominantly sequential languages, their exploration in a concurrent, message-passing setting has been more confined to typed process calculi [34, 35, 17, 25, 41, 68, 55] and multiparty session types [13, 11, 14, 12]. Only recently, IFC has been adopted for session types based on linear logic [20, 5]. The resulting type systems exploit the strong guarantees arising from linear logic, which in particular curtail the network of processes arising at runtime to a tree structure. However, many real-world application scenarios are precluded from an insistence on a tree structure, instead requiring support of *cyclic process networks*. Session type systems [6, 7, 19, 29, 30] that allow for cyclic process networks increase expressivity while remaining rooted in linear logic.

This paper scales IFC to cyclic process networks and contributes an IFC type system for an asynchronous π -calculus with linear session types. To prove that well-typed processes in the resulting language enjoy noninterference, we develop a novel logical relation. Our development was challenged by the possibility of *deadlocks* that can arise in cyclic process networks and that constitute another form of side channel. To rule out side-channel attacks due to deadlocks, we introduce the notion of *deadlock-sensitive noninterference (DSNI)*, which only equates a deadlocking program with another deadlocking one. Using our logical relation we prove that well-typed processes in our language enjoy DSNI (fundamental theorem).

Cyclic process networks also turn out to be beneficial for IFC, as they permit secure programs that are rejected by existing IFC type systems for linear session types [20, 5]. These are programs that exploit the possibility of setting up several channels – rather than just one channel – between two processes, to separate low-security from high-security communication.

Contributions. Our contributions are threefold:

1. An IFC session type system for an asynchronous π -calculus with support for cyclic process networks (Sec. 3), that satisfies protocol fidelity and communication safety (Thm. 3.12).
2. A logical relation that induces an equivalence between typed processes (Sec. 4), defining our notion of DSNI (Def. 4.9).
3. The main result that well typedness implies DSNI (Thm. 5.1 in Sec. 5), following from the fundamental theorem (Thm. 5.7).

Outline. In addition to the above contributions, Sec. 2 gently introduces the key ideas behind the developments in this paper, Sec. 6 discusses related work, and Sec. 7 concludes the paper. Important proofs are detailed in part; remaining details, proofs, and auxiliary definitions are given in the extended version of this paper [27].

2 Key Ideas

In this section, we discuss the key ideas behind our contributions.

2.1 Cyclic Process Networks Afford Flexible Information Flow Control

We motivate our developments through a high-level example, focusing on how cyclicity in process networks improves over prior works by increasing the flexibility of information flow control to support more realistic scenarios.

Collaborating governments. Consider two governments that want to collaborate on scientific and intelligence efforts. Clearly, the interactions between a government and an intelligence agency is confidential, whereas interactions between a government and the scientific community is not; intelligence may not leak to the scientific community, where there may be spies.

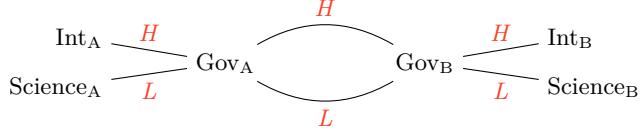
We make this more precise by establishing that information can be of *High* or *Low* confidentiality, and that communication channels can be of *High* or *Low* security. Clearly, information of *Low* confidentiality can be transmitted over *High*-security channels, but not vice versa. We identify our two governments as $X \in \{A, B\}$, each with departments (processes) $\text{Gov}_X, \text{Int}_X, \text{Science}_X$. We consider three scenarios, each of which connects these processes to form different process networks.

Scenario 1: No cyclicity. In Fig. 1a, the governments have only one channel to communicate on. Lines between departments denote communication channels, and their annotations indicate security levels. In this scenario, no processes are cyclically connected.

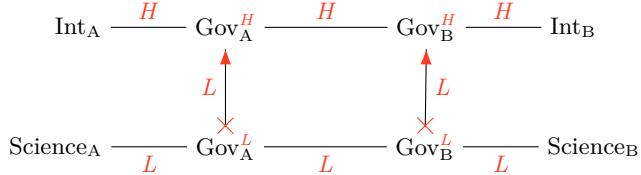
Notice that the channel connecting the two governments does not have a security level assigned. If we insist on intelligence exchange (i.e., on exchanging *High*-confidentiality information), this channel must be of *High* security. However, this inhibits scientific exchange: when a government receives information on a *High*-security channel, it cannot guarantee that the information is of *Low* confidentiality, so it cannot share it with its scientific department over a *Low*-security channel. Hence, the single channel of communication between the governments is unrealistic.



(a) Scenario 1: No cyclicity.



(b) Scenario 2: Doubly connected governments.



(c) Scenario 3: Extended cyclicity for more flexible IFC.

Figure 1 Collaborating governments: three scenarios.

Scenario 2: Doubly-connected governments. In Fig. 1b, we attempt to remedy this problem by adding a second channel of *Low* security between the governments. Since the governments are connected on two separate channels, they are cyclically connected.

Now the governments can exchange scientific information and share this with their intelligence agencies. However, it is conceivable that a government makes decisions about which scientific information to share based on intelligence information. A clever spy may then be able to infer intelligence information from scientific information, *indirectly*. Hence, once a government receives intelligence information, it should refrain from sharing scientific information. Clearly, this scenario is still not realistic.

Scenario 3: Extended cyclicity for more flexible IFC. In Fig. 1c, we split our governments into *High*- and *Low*-confidentiality departments. The *High*-confidentiality departments share intelligence information, and the *Low*-confidentiality departments share scientific information. Crucially, the *Low*-confidentiality department can share information with the *High*-confidentiality department, but not vice versa.

2.2 Threats to Noninterference due to Deadlocks

When process networks contain cyclic connections, there is a risk of *deadlocked* communication. For example, consider again the process network in Fig. 1b. Let us refer to the *High*-security channel between the governments as *h*, and the *Low*-security channel between the governments as *l*. Suppose the two governments are implemented as follows (in pseudocode):

$\text{Gov}_A := \text{receive on } h; \text{send on } l$

$\text{Gov}_B := \text{receive on } l; \text{send on } h$

The communication between the governments is deadlocked: each government is waiting to receive from the other, but the corresponding sends are blocked.

In process networks without cyclicity, this kind of deadlock does not occur. There are ways to prevent them through typing (cf., e.g., [19, 30, 7]), but the possible occurrence of deadlock introduced by cyclicity is realistic and a possible threat to noninterference. To see how, consider another pseudocode implementation, where s_A refers to the *Low*-security channel between Gov_A and Science_A :

$$\text{Gov}_A := \text{receive } x \text{ on } h; \text{if } x == \text{true} \text{ then send on } s_A \text{ else deadlock} \quad (1)$$

In this scenario, a spy monitoring the information exchanged on s_A is indirectly able to infer *High*-confidentiality information: if information is sent on s_A , then the spy knows for sure that the value of x is true. This is why we are after IFC for noninterference that is *deadlock sensitive*.

2.3 IFC Type System in a Nutshell

In this paper, we implement IFC similarly as in previous works: by enriching a session type system with IFC annotations and requirements. We build on the session-typed asynchronous variant of the π -calculus of Van den Heuvel and Pérez [29, 30] (stripped from the “priority” mechanisms that rule out deadlock).

As anticipated in Sec. 2.1, channels are appointed *maximum-secrecy levels* (secrecy levels, for short), indicating the maximum secrecy of messages that can be sent on channels securely. For example, in Fig. 1b, the channel between Int_A and Gov_A has a secrecy level of *High*, while the channel between Science_A and Gov_A has a secrecy level of *Low*. This indicates that a spy with low-level security clearance can observe the messages sent on channels of secrecy level *Low* but not *High*. A partial order on these secrecy levels forms a *secrecy lattice*; for example, $L \sqsubseteq H$ (*Low* is lower than *High*).

As processes receive messages on channels, they learn “secrets”, possibly influencing the information sent in future messages (referred to in the literature as *flow sensitivity* [57]). Key in our IFC is thus that it is forbidden to send messages on a channel if the level of secrecy learned so far exceeds the secrecy level of the channel. To ensure this, our type system assigns to each process a *running secrecy* that increases as the process receives higher-secrecy-level information. As processes evolve, the secrecy levels of their channels do not change, whereas their running securties do.

For example, in Fig. 1b, suppose Gov_A starts with a *Low* running secrecy, thus being able to send messages to both Int_A and Science_A . After receiving a message from Int_A , the running secrecy of Gov_A becomes *High*: the secrecy level of the channel between them is *High*. Hence, after this message, Gov_A can no longer send messages to Science_A .

Finally, we need to address how our IFC handles deadlock sensitivity, as introduced in Sec. 2.2. It turns out that it is sufficient to rely on running securties and their dynamics as described above. To see how, consider again the implementation of Gov_A in (1). Assuming it starts with *Low* running secrecy, the process receives on a *High*-security channel, so its running secrecy becomes *High*. Our IFC then disallows the process from sending on the *Low* security channel. Hence, this example would be considered ill typed in our type system.

2.4 Logical Relation for DSNI in a Nutshell

Let us be more precise in what we mean by DSNI. A process may have a number of “unconnected” channels. By connecting these channels to other processes, we create a *context* in which to run the process. We refer to the channels connecting the process to its context as the *interface*. For example, in Fig. 1b, Gov_A can be considered as a standalone process, with the rest of the processes being a potential context for it. The interface is then the four channels connecting Gov_A to the other processes.

With DSNI, we assume the existence of an “attacker”, a more precise definition of the “spy” mentioned in Sec. 2.1. This attacker knows the specification of our process, and has the ability to observe messages from and to the process over *observable channels*: channels in the interface that have secrecy levels up to a given secrecy level ξ . Moreover, the attacker cannot measure time but can observe the relative order in which messages are sent through different channels. By running our process in different contexts and observing the messages on observable channels, the attacker may be able to use its knowledge of the process’ specification to infer information about messages on unobservable channels. As such, noninterference means that the attacker is not able to do so; in our case, we are after DSNI, because we do not want the attacker to infer information from deadlocks either.

In this paper, we define DSNI as an *equivalence* between the behavior on observable channels of the same process in different contexts. This equivalence is defined by means of a *logical relation*. The relation scrutinizes messages from and to the process on observable channels, and “ignores” messages on unobservable channels. Our main result is that well typedness implies DSNI (Thm. 5.1).

2.5 Technical Challenges

The subsequent sections first introduce our process language and then develop an IFC type system for that language and state and prove DSNI using a logical relation. These sections are naturally quite technical. To bridge the divide, we briefly survey here the main challenges our development had to overcome.

Asynchronous communication. Our process language is an *asynchronous π -calculus* with linear session types, based on Van den Heuvel and Pérez’s Asynchronous Priority-based Classical Processes (APCP) [29, 30], but without recursion and “priority” mechanisms (which prevent deadlocks). As in the asynchronous π -calculus, outputs in our calculus do not have any continuations but are atomic processes composed in parallel with other processes. To model session sequencing, a process must adopt a *continuation-passing* style, in the sense that an output not only comprises a message but also a continuation channel.

When continuation channels are part of messages exchanged over an observable channel in the interface between a process and its context, the question comes up whether the continuation channel becomes observable as well. The natural impulse might be to consider them observable too. For sure this is the right choice in linear session-typed process calculi that confine process networks to trees [20, 5], guaranteeing that the continuation channel sent as part of the message resides within the sending process itself. However, due to the possibility of cycles in our setting, a continuation channel sent as part of a message may actually reside within the context outside the sending process. As a result, the logical relation has to consider the binding structure of the process and the context when determining observability of continuation channels. We detail this case analysis in Sec. 4 when we introduce the logical relation, with a pictorial illustration in Fig. 7.

Observable deadlocks. Deadlock-sensitive noninterference (DSNI) provides a very strong notion of noninterference in that it equates a deadlocking process only with another deadlocking one (as opposed to an arbitrary one). As a result, it prevents leakage through deadlocks, a side channel similar to the termination channel. DSNI is asserted by the definition of the logical relation and challenges the proof of Thm. 5.1, stating that well typedness implies DSNI. Thm. 5.1 is proved a generalized *fundamental theorem* (Thm. 5.7), which asserts that all executions of a process, if well typed, are related by the logical relation, up to the secrecy

level ξ of the observer. Because this theorem relates two different processes, but with the same observable behavior (where deadlocks are observable), the proof must maintain a tight correspondence between the two processes. This correspondence is achieved by employing the notion of *relevant nodes* (Def. 5.5), which are the parts of a process that can have observable outcomes either directly (by sending a message over the interface) or indirectly (by initiating a chain of messages ending with an observable one), and asserting that the relevant nodes of both processes are indistinguishable (up to structural congruence). Our notion of relevant nodes is inspired by Derakhshan et al. [20], but accounts for cyclic process networks.

Structural congruence and alpha equivalence. Our logical relation makes use of structural congruence to single out the action in a process producing an observable message. Because structural congruence permits alpha renaming, process relatedness must account for alpha-equivalence classes. As usual, proofs require a careful treatment of alpha renaming, which additionally becomes more nuanced by the existence of binders for observable names in contexts. This treatment becomes especially apparent in the so-called *catch-up* lemma (Lem. 5.10), a lemma used in the proof of the fundamental theorem to assert that two observably equivalent processes can “catch up” on each other’s unobservable reductions.

3 Linear Session Types for Information Flow Control

In this section, we define our information flow control (IFC) type system. We first introduce our process language (an asynchronous π -calculus) along with a linear session-type system in Sec. 3.1. Then, we enrich the type system with IFC in Sec. 3.2. In Sec. 3.3, we prove that well-typed processes enjoy communication safety and protocol fidelity as corollaries of a type-preservation result. As we will see in Sec. 5, well typedness in the resulting IFC type system implies noninterference.

3.1 Process Language: Syntax, Semantics, and Types

Our process language is an asynchronous π -calculus, where parallel subprocesses communicate on connected channels. To be precise, we adapt the non-recursive fragment of Van den Heuvel and Pérez’s Asynchronous Priority-based Classical Processes (APCP) [29, 30] by removing their “priority” mechanisms that prevent deadlock and adding our IFC.

Syntax. The syntactic elements of our language are typeset in a black and non-italic font. In our language, channels have two distinct endpoints, denoted a, b, c, \dots, x, y, z and further referred to as *names*. By design, all names are used linearly, meaning that they are used for a communication exactly once.

► **Definition 3.1** (Syntax). Processes P, Q, R, \dots are defined by the following syntax:

$$P, Q, R, \dots ::= 0 \mid (P \mid Q) \mid (\nu xy)P \mid x[] \mid x(); P \mid x[b] \triangleleft j \mid x(z) \triangleright \{i : P_i\}_{i \in I} \mid x[a, b] \mid x(y, z); P$$

We write $P\{x/y\}$ to denote the capture-avoiding substitution of y for x in P . Process 0 denotes inaction. In $(P \mid Q)$, processes P and Q run in parallel; we often omit the parentheses. Restriction $(\nu xy)P$ binds x and y in P to form a channel, enabling communication.

Process $x[]$ closes the channel to which x belongs, and $x(); P$ waits for the channel to close before continuing as P . Selection $x[b] \triangleleft j$ sends the label j over x along with a name b ; we refer to b as the selection’s *continuation*, as it provides a means to continue communicating after the selection. Branch $x(z) \triangleright \{i : P_i\}_{i \in I}$ waits to receive on x a label $j \in I$ along with

a continuation b before continuing as $P_j\{b/z\}$; this binds z in each P_i . Send $x[a, b]$ sends names a and b over x ; we typically refer to a and b as the send's payload and continuation, respectively, but there is no technical distinction between them. Receive $x(y, z); P$ waits to receive on x two names a and b before continuing as $P\{a/y, b/z\}$; this binds y and z in P . All names in a process are free unless bound as described above; we write $fn(P)$ to denote the set of free names of P .

► **Example 3.2.** To illustrate process syntax, we further develop the example introduced in Sec. 2.1. We develop two simple accounts of Gov_A : one where information flow is secure, and one where it is not.

In the first scenario, Gov_A^L passes a research outcome (oc) to Gov_A^H , which determines a command for Int_A :

$$\begin{aligned}\text{Gov}_A^L &:= (\nu a_H^{1'} a_H^1)(a_H[a_H^{1'}] \triangleleft oc_2 | a_H^1[])) \\ \text{Gov}_A^H &:= a_L(a_L^1) \triangleright \left\{ \begin{array}{l} oc_1 : (\nu a_I^{1'} a_I^1)(a_I[a_I^{1'}] \triangleleft act | a_L^1(); a_I^1[]), \\ oc_2 : (\nu a_I^{1'} a_I^1)(a_I[a_I^{1'}] \triangleleft wait | a_L^1(); a_I^1[])) \end{array} \right\} \\ \text{Int}_A &:= i_A(i_A^1) \triangleright \{act : i_A^1(); 0, wait : i_A^1(); 0\} \\ A_{\text{secure}} &:= (\nu a_H a_L)(\nu a_I i_A)(\text{Gov}_A^L | \text{Gov}_A^H | \text{Int}_A)\end{aligned}$$

In the second scenario, Gov_A^H receives intelligence (int) from Int_A and shares the information (inf) with Gov_A^L :

$$\begin{aligned}\text{Int}_A &:= (\nu i_A^{1'} i_A^1)(i_A[i_A^{1'}] \triangleleft int_1 | i_A^1[]) \\ \text{Gov}_A^H &:= a_I(a_I^1) \triangleright \left\{ \begin{array}{l} int_1 : (\nu a_L^{1'} a_L^1)(a_L[a_L^{1'}] \triangleleft inf_1 | a_I^1(); a_L^1[]), \\ int_2 : (\nu a_L^{1'} a_L^1)(a_L[a_L^{1'}] \triangleleft inf_2 | a_I^1(); a_L^1[])) \end{array} \right\} \\ \text{Gov}_A^L &:= a_H(a_H^1) \triangleright \{inf_1 : a_H^1(); 0, inf_2 : a_H^1(); 0\} \\ A_{\text{insecure}} &:= (\nu a_H a_L)(\nu a_I i_A)(\text{Gov}_A^L | \text{Gov}_A^H | \text{Int}_A)\end{aligned}$$

Variants of the π -calculus often include the forwarder process $[x \leftrightarrow y]$ which forwards any communications between x and y by fusing x and y . Here, we choose to omit forwarders for a smoother definition of our logical relation; they can be added as syntactic sugar using *identity expansion* (cf. the extended paper).

Semantics. The dynamics of our language is defined in terms of a reduction semantics, where each step represents the synchronization of complementary communications on the two endpoints of a channel. As usual, reduction relies on structural congruence, which restructures processes without affecting channel connections and the order of communications.

► **Definition 3.3 (Reduction Semantics).** Structural congruence is the least congruence on the syntax of processes (i.e., closed under arbitrary process contexts), denoted $P \equiv Q$, induced by the axioms in Fig. 2 (top).

Reduction is a binary relation on processes, denoted $P \rightarrow Q$, defined by the rules in Fig. 2 (bottom). We write $P \not\rightarrow$ to denote that there is no Q such that $P \rightarrow Q$.

Rule [SC-ALPHA] allows alpha conversion, i.e., renaming bound names. Rule [SC-PAR-NIL] defines 0 as the unit of parallel composition, and Rules [SC-PAR-SYMM] and [SC-PAR-ASSOC] define parallel composition as symmetric and associative, respectively. Rules [SC-RES-SYMM] and [SC-RES-ASSOC] define symmetry and associativity of restriction, respectively. Rule [SC-RES-COMM] defines commutativity of restriction, as long as this does not capture or free any names; this is often referred to as *scope extrusion*.

Structural congruence ($P \equiv Q$):

$$\begin{array}{c}
 \frac{[\text{SC-ALPHA}]}{P \equiv_\alpha Q} \quad \frac{[\text{SC-PAR-NIL}]}{P \mid 0 \equiv P} \quad \frac{[\text{SC-PAR-SYMM}]}{P \mid Q \equiv Q \mid P} \quad \frac{[\text{SC-PAR-ASSOC}]}{(P \mid Q) \mid R \equiv P \mid (Q \mid R)} \quad \frac{[\text{SC-RES-SYMM}]}{(\nu xy)P \equiv (\nu yx)P} \\
 \\
 \frac{[\text{SC-RES-ASSOC}]}{(\nu xy)(\nu zw)P \equiv (\nu zw)(\nu xy)P} \quad \frac{[\text{SC-RES-COMM}]}{x, y \notin \text{fn}(Q)} \quad \frac{}{(\nu xy)(P \mid Q) \equiv (\nu xy)P \mid Q}
 \end{array}$$

Reduction ($P \rightarrow Q$):

$$\begin{array}{c}
 \frac{[\text{RED-CLOSE-WAIT}]}{(\nu xy)(x[] \mid y(); P) \rightarrow P} \quad \frac{[\text{RED-SEL-BRA}]}{j \in I} \quad \frac{[\text{RED-SC}]}{P \equiv P' \quad P' \rightarrow Q' \quad Q' \equiv Q} \quad \frac{[\text{RED-PAR}]}{P \rightarrow P'} \\
 \\
 \frac{[\text{RED-SEND-RECV}]}{(\nu xy)(x[a, b] \mid y(z, w); Q) \rightarrow Q\{a/z, b/w\}} \quad \frac{P \rightarrow Q}{P \rightarrow Q} \quad \frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q} \\
 \\
 \frac{[\text{RED-RES}]}{P \rightarrow P'} \quad \frac{(\nu xy)P \rightarrow (\nu xy)P'}{(\nu xy)P \rightarrow (\nu xy)P'}
 \end{array}$$

■ **Figure 2** Structural congruence (top) and reduction (bottom); cf. Def. 3.3.

Rules [RED-CLOSE-WAIT], [RED-SEL-BRA], and [RED-SEND-RECV] define synchronizations of complementary communications on names connected by restriction; these rules formalize the behavior described below Def. 3.1. Rules [RED-SC], [RED-PAR], and [RED-RES] close reduction under structural congruence, parallel composition, and restriction, respectively.

► **Example 3.4.** We illustrate process semantics on A_{secure} defined in Example 3.2. We have

$$\begin{aligned}
 A_{\text{secure}} &= (\nu a_H a_L)(\nu a_I i_A)(\text{Gov}_A^L \mid \text{Gov}_A^H \mid \text{Int}_A) \\
 &\equiv (\nu a_I i_A)((\nu a_H^1 a_H^1)((\nu a_H a_L)(a_H[a_H^1] \triangleleft \text{oc}_2 \mid a_L(a_L^1) \triangleright \{\dots\}) \mid a_H^1[]) \mid \text{Int}_A) \\
 &\rightarrow (\nu a_I i_A)((\nu a_H^1 a_H^1)((\nu a_I^1 a_I^1)(a_I[a_I^1] \triangleleft \text{wait} \mid a_H^1() \mid a_I^1[]) \mid a_H^1[]) \mid \text{Int}_A),
 \end{aligned}$$

from where asynchronous communication enables further communication between a_H^1 and a_H^1 or between a_I and i_A ; for example,

$$\rightarrow (\nu a_I i_A)((\nu a_I^1 a_I^1)(a_I[a_I^1] \triangleleft \text{wait} \mid a_I^1[]) \mid \text{Int}_A).$$

Types. We use linear session types to “tame” our processes. The system we use is derived from classical linear logic, so types are expressed as linear-logic propositions¹; they are typeset in a blue and sans-serif font.

► **Definition 3.5 (Types).** Types A, B, C, \dots are defined by the following syntax:

$$A, B, C, \dots ::= 1 \mid \perp \mid \oplus\{i : A\}_{i \in I} \mid \&\{i : A\}_{i \in I} \mid A \otimes B \mid A \wp B$$

¹ This choice is usually motivated as it comes with deadlock freedom, but we have two different reasons: (1) it allows for direct compatibility with session-type systems for deadlock freedom, and (2) a logical basis gives us a very clean and well-behaved linear session type system, which we can carefully extend to serve our goals (here, guaranteeing noninterference by typing).

$$\begin{array}{c}
 \frac{[\text{TYP-INACT}]}{\Omega \vdash 0 @ d :: \emptyset} \quad \frac{[\text{TYP-PAR}]}{\Omega \Vdash d \sqsubseteq d'_1 \sqcap d'_2 \quad \Omega \vdash P @ d'_1 :: \Gamma \quad \Omega \vdash Q @ d'_2 :: \Delta}{\Omega \vdash P | Q @ d :: \Gamma, \Delta} \\
 \\
 \frac{[\text{TYP-RES}]}{\Omega \vdash P @ d :: \Gamma, x : A[c], y : A^\perp[c]} \quad \frac{[\text{TYP-CLOSE}]}{\Omega \Vdash d \sqsubseteq c}{\Omega \vdash x[] @ d :: x : 1[c]} \quad \frac{[\text{TYP-WAIT}]}{\Omega \Vdash d' = d \sqcup c \quad \Omega \vdash P @ d' :: \Gamma}{\Omega \vdash x(); P @ d :: \Gamma, x : \perp[c]} \\
 \\
 \frac{[\text{TYP-SEL}]}{\Omega \Vdash d \sqsubseteq c \quad j \in I}{\Omega \vdash x[b] \triangleleft j @ d :: x : \oplus\{i : A_i\}_{i \in I}[c], b : A_j^\perp[c]} \quad \frac{[\text{TYP-BRA}]}{\Omega \Vdash d' = d \sqcup c \quad \forall i \in I. \Omega \vdash P_i @ d' :: \Gamma, z : A_i[c]}{\Omega \vdash x(z) \triangleright \{i : P_i\}_{i \in I} @ d :: \Gamma, x : \&\{i : A_i\}_{i \in I}[c]} \\
 \\
 \frac{[\text{TYP-SEND}]}{\Omega \Vdash d \sqsubseteq c}{\Omega \vdash x[a, b] @ d :: x : A \otimes B[c], a : A^\perp[c], b : B^\perp[c]} \\
 \\
 \frac{[\text{TYP-RECV}]}{\Omega \Vdash d' = d \sqcup c \quad \Omega \vdash P @ d' :: \Gamma, y : A[c], z : B[c]}{\Omega \vdash x(y, z); P @ d :: \Gamma, x : A \wp B[c]}
 \end{array}$$

Figure 3 Typing rules; cf. Def. 3.6.

Duality is a unary operation on types, denoted A^\perp , defined as follows:

$$\begin{array}{lll}
 1^\perp := \perp & \oplus\{i : A_i\}_{i \in I}^\perp := \&\{i : A_i^\perp\}_{i \in I} & A \otimes B^\perp := A^\perp \wp B^\perp \\
 \perp^\perp := 1 & \&\{i : A_i\}_{i \in I}^\perp := \oplus\{i : A_i^\perp\}_{i \in I} & A \wp B^\perp := A^\perp \otimes B^\perp
 \end{array}$$

Type 1 is associated with names that close channels, and \perp with names that wait for channels to close. Types $\oplus\{i : A_i\}_{i \in I}$ and $\&\{i : A_i\}_{i \in I}$ are associated with names that make and expect labeled selections, respectively; given $j \in I$, A_j is the type of the continuation after j has been selected/received. Types $A \otimes B$ and $A \wp B$ are associated with names that send and receive, respectively; A and B are the types of the payload and continuation afterwards.

Duality is a key component of session types, as it defines precisely what is meant by complementary behavior; for example, $1^\perp = \perp$ is complementary to 1 . Clearly, duality is an involution (i.e., $(A^\perp)^\perp = A$).

Our type system is defined as a sequent calculus. In the following, ignore the annotations in *red and italic*; these annotations are for IFC, explained in Sec. 3.2.

► **Definition 3.6** (Type System). Typing contexts Γ, Δ, \dots are defined by the following syntax:

$$\Gamma, \Delta, \dots ::= \emptyset \mid \Gamma, x : A[c]$$

Typing judgments are denoted $\Omega \vdash P @ d :: \Gamma$. They are derived using the rules in Fig. 3.

Typing contexts are thus sets of types assigned to names; the type system allows implicitly reordering these assignments in typing contexts. Whenever we write Γ, Δ , we assume that the sets of names appearing in Γ and Δ are disjoint.

Rule [TYP-INACT] types inaction under empty context. Rule [TYP-PAR] types parallel composition by splitting the typing context into disjoint parts, one for each parallel process. Rule [TYP-RES] types restriction by requiring the connected names to be dually typed. Rule [TYP-CLOSE] types a close with only its subject in the context. Dually, Rule [TYP-WAIT]

types a wait by removing its subject from the context of the continuation. Rule [TYP-SEL] types a selection; note that the continuation is typed dually to the continuation type of the selection itself, as this name will be received by a corresponding branch and used for further communications there. Dually, Rule [TYP-BRA] types a branch; it requires every continuation to be typed with the same context besides the type of the continuation name. Rule [TYP-SEND] types a send; the payload and continuation are typed dually, similar to the continuation in Rule [TYP-SEL]. Dually, Rule [TYP-RECV] types a receive.

► **Example 3.7.** To illustrate process typing, we type the secure variant of Gov_A^H introduced in Example 3.2 as follows. We omit the *red and italic* IFC annotations entirely, as well as the typing of the oc_2 branch which is analogous to the oc_1 branch.

$$\begin{array}{c}
 \frac{}{\vdash a_I[a_I^{1'}]\triangleleft\text{act}} \quad \frac{\vdash a_I^1[]::a_I^1:1}{\vdash a_L^1();a_I^1[]} \quad \vdots \\
 \text{[TYP-SEL]} \qquad \text{[TYP-CLOSE]} \qquad \text{[TYP-WAIT]} \\
 ::a_I:\oplus\{act:1,wait:1\},a_I^{1'}:\perp \qquad ::a_L^1:\perp,a_I^1:1 \qquad ::a_L^1:\perp,a_I:\oplus\{act:1,wait:1\} \\
 \frac{}{\vdash a_I[a_I^{1'}]\triangleleft\text{act}|a_L^1();a_I^1[]}\qquad \frac{}{\vdash a_L^1:\perp,a_I:\oplus\{act:1,wait:1\},a_I^{1'}:\perp,a_I^1:1} \quad \text{[TYP-PAR]} \\
 ::a_L^1:\perp,a_I:\oplus\{act:1,wait:1\},a_I^{1'}:\perp,a_I^1:1 \qquad \frac{\vdash (\nu a_I^{1'}a_I^1)(a_I[a_I^{1'}]\triangleleft\text{act}|a_L^1();a_I^1[])}{::a_L^1:\perp,a_I:\oplus\{act:1,wait:1\}} \quad \text{[TYP-RES]} \\
 \frac{}{\vdash a_L(a_L^1)\triangleright\left\{\begin{array}{l} oc_1:(\nu a_I^{1'}a_I^1)(a_I[a_I^{1'}]\triangleleft\text{act}|a_L^1();a_I^1[]), \\ oc_2:(\nu a_I^{1'}a_I^1)(a_I[a_I^{1'}]\triangleleft\text{wait}|a_L^1();a_I^1[]) \end{array}\right\}::a_L:\&\{oc_1:\perp,oc_2:\perp\},a_I:\oplus\{act:1,wait:1\}} \quad \text{[TYP-BRA]}
 \end{array}$$

3.2 Information Flow Control

We now enrich the type system presented thus far with IFC, such that well typedness guarantees noninterference (Sec. 5). That is, we introduce and explain the annotations in *red and italic* in Fig. 3, and formalize the intuitions given in Sec. 2.3.

We use c, d, \dots to denote secrecy levels. The relation between secrecy levels is denoted $c \sqsubseteq d$ (c is at most as secret as d), forming a lattice Ω . We write $\Omega \Vdash \phi$ to denote that the relation ϕ between secrecy levels holds within Ω . The least upper bound (join) and greatest lower bound (meet) are denoted $c \sqcup d$ and $c \sqcap d$, respectively.

Every name is assigned a secrecy level, denoted in typing contexts using square brackets after the name's type, as in $x:A[c]$. To remember the level of secrecy of the messages received by a process, we annotate the process in typing judgments with a *running secrecy*, denoted $P @ d$. Given the running secrecy d of the process before an input and the secrecy level c of the input's subject, the input updates the running secrecy to the join $d \sqcup c$. When a process then performs an output, to make sure that the name is secured for handling the secrecy level of the outgoing message, our IFC requires that its running secrecy is not higher than that of the output's subject name.

We make these intuitions precise by discussing the IFC annotations on each typing rule in Fig. 3. Since Rule [TYP-INACT] does not involve communication, no secrecy checks are necessary. Rule [TYP-CLOSE] types an output, so it requires that the running secrecy d of the close is at most the secrecy level c of the closed name ($d \sqsubseteq c$): the information received so far (the running secrecy) is not more secret than the closed name. Rules [TYP-SEL] and [TYP-SEND] also type outputs, so their checks are similar. Rule [TYP-WAIT] types an input, so it sets the running secrecy d' of the continuation of the wait to the least upper bound of the running secrecy d before the wait and the secrecy level c of the name of the wait ($d' = d \sqcup c$; i.e., to the least secrecy level that is at least as high as both involved secrecyies).

Rules [TYP-BRA] and [TYP-RECV] also type inputs, so they update running seccrecies similarly. Rule [TYP-PAR] combines the running seccrecies d'_1 and d'_2 of the parallel processes by taking a seccrecy level d that is at most the greatest lower bound of d'_1 and d'_2 ($d \sqsubseteq d'_1 \sqcap d'_2$); this way, the parallel composition has a running seccrecy of at most the least common seccrecy level of its components, ensuring that each process in the composition has at least as much information as the parallel composition itself. Rule [TYP-RES] requires that the seccrecies of the connected names coincide, ensuring that seccrecy checks are consistent on both names of the created channel. Note that channels in typing contexts may have different seccrecy levels. However, typing rules enforce that the channels in the same session (e.g., a send and its continuation) have the same seccrecy levels. For example, Rule [TYP-SEL] ensures that channel x and its continuation b are both of the same seccrecy level c .

► **Example 3.8.** To illustrate IFC in our type system, we consider again the typing of Gov_A^H from Examples 3.2 and 3.7. We repeat the typing derivation and include IFC annotations, but omit processes and types to save space. Let Ω be the lattice with the only relation $L \sqsubseteq H$.

$$\frac{\frac{\frac{L \sqsubseteq H}{\Omega \vdash @L :: a_I : [H], a_I^1 : [H]} \text{[TYP-SEL]} \quad \frac{\frac{L \sqsubseteq H}{\Omega \vdash @L \sqcup L = L :: a_I^1 : [H]} \text{[TYP-CLOSE]} \quad \frac{\frac{L \sqsubseteq H}{\Omega \vdash @L :: a_L^1 : [L], a_I^1 : [H]} \text{[TYP-WAIT]}}{\Omega \vdash @L :: a_L^1 : [L], a_I^1 : [H]} \text{[TYP-PAR]}}{\Omega \vdash @L :: a_L^1 : [L], a_I : [H], a_I^1 : [H], a_I^1 : [H]} \text{[TYP-RES]}}{\Omega \vdash @L \sqcup L = L :: a_L^1 : [L], a_I : [H]} \text{[TYP-BRA]}$$

Hence, Gov_A^H is considered secure in our type system, and as we will show in Sec. 5 this means that noninterference holds for this process.

However, the initial assignment of maximum seccrecies to endpoints is chosen by the user. Well typedness and thus noninterference depends on this initial choice. To illustrate, reconsider the derivation above but now swapping the initial maximum seccrecies:

$$\frac{\frac{\frac{H \not\sqsubseteq L}{\Omega \vdash @H :: a_I : [L], a_I^1 : [L]} \text{[TYP-SEL]} \quad \frac{\frac{H \not\sqsubseteq L}{\Omega \vdash @H \sqcup H = H :: a_I^1 : [L]} \text{[TYP-CLOSE]} \quad \frac{\frac{H \not\sqsubseteq L}{\Omega \vdash @H :: a_L^1 : [H], a_I^1 : [L]} \text{[TYP-WAIT]}}{\Omega \vdash @H :: a_L^1 : [H], a_I^1 : [L]} \text{[TYP-PAR]}}{\Omega \vdash @H :: a_L^1 : [H], a_I : [L], a_I^1 : [L], a_I^1 : [L]} \text{[TYP-RES]}}{\Omega \vdash @L \sqcup H = H :: a_L^1 : [H], a_I : [L]} \text{[TYP-BRA]}$$

This time, Gov_A^H is not well typed: the IFC requirements of Rules [TYP-SEL] and [TYP-CLOSE] do not hold.

3.3 Type Preservation

Our type system guarantees by well typedness the usual correctness properties: *session fidelity* and *communication safety*. The former states that a process correctly implements the session types assigned to its names, and the latter that no communication mismatches take place (such as simultaneous outputs on both names of a channel).

Both these properties follow directly from *type preservation*: well typedness is preserved across structural congruences (subject congruence; Thm. 3.11) and reduction (subject reduction; Thm. 3.12). These results rely on two lemmas:

- Lem. 3.9 states that names that are not free in a process are not assigned in the typing of the process.
- Lem. 3.10 states that substitution in a process is reflected in its typing.

► **Lemma 3.9.** *Given $\Omega \vdash P @ d :: \Gamma$, if $x \notin \text{fn}(P)$, then $x \notin \text{dom}(\Gamma)$.*

► **Lemma 3.10 (Substitution).** *Given $\Omega \vdash P @ d :: \Gamma, x : A[c]$, we have*

$$\Omega \vdash P\{y/x\} @ d :: \Gamma, y : A[c].$$

► **Theorem 3.11 (Subject Congruence).** *If $\Omega \vdash P @ d :: \Gamma$ and $P \equiv Q$ for some Q , then $\Omega \vdash Q @ d :: \Gamma$.*

Proof. By induction on the derivation of $P \equiv Q$. The inductive cases correspond to closure under arbitrary process contexts in Def. 3.1; these cases follow from the IH straightforwardly. The base cases correspond to the seven rules in Fig. 2 (top). In each case, we apply inversion on the typing of P to derive the typing of Q , and vice versa, with straightforward reasoning about running seccuries. The only interesting case is Rule [SC-PAR-ASSOC] $((P | Q) | R \equiv P | (Q | R))$, where we derive the running secrecy of $Q | R$ from that of $P | Q$, and vice versa. The full proof is in the extended paper. ◀

The following theorem states that (i) reduction preserves the well typedness of processes, and (ii) the running secrecy of processes may either stay the same or increase during reduction. This implies that a process never forgets the secrets it has learned, but it may learn more secrets as it reduces.

► **Theorem 3.12 (Subject Reduction).** *If $\Omega \vdash P @ d :: \Gamma$ and $P \rightarrow Q$ for some Q , then $\Omega \vdash Q @ d' :: \Gamma$ for some d' such that $\Omega \Vdash d \sqsubseteq d'$.*

Proof. By induction on the derivation of $P \rightarrow Q$. The cases correspond to the reduction rules in Fig. 2 (bottom). In each case, we apply inversion on the typing of P to derive the typing of Q . The full proof is in the extended paper; here, we show the interesting case of Rule [RED-SEND-RECV]: $(\nu xy)(x[a, b] | y(z, w); P) \rightarrow P\{a/z, b/w\}$. Given

$$\Omega \Vdash d \sqsubseteq d'_1 \sqcap d'_2, \tag{2}$$

$$\frac{\Omega \Vdash d'_1 \sqsubseteq c}{\Omega \vdash x[a, b] @ d'_1 :: x : A \otimes B[c], a : A^\perp[c], b : B^\perp[c]} \text{ [TYP-SEND]}, \tag{3}$$

$$\Omega \Vdash d''_2 = d'_2 \sqcup c, \tag{4}$$

we have

$$\frac{\begin{array}{c} (4) \quad \Omega \vdash P @ d''_2 :: \Gamma, z : A^\perp[c], w : B^\perp[c] \\ (2) \quad \frac{(3) \quad \Omega \vdash y(z, w); P @ d'_2 :: \Gamma, y : A^\perp \wp B^\perp[c]}{\Omega \vdash x[a, b] | y(z, w); P @ d :: \Gamma, x : A \otimes B[c], y : A^\perp \wp B^\perp[c], a : A^\perp[c], b : B^\perp[c]} \text{ [TYP-PAR]} \end{array}}{\Omega \vdash (\nu xy)(x[a, b] | y(z, w); P) @ d :: \Gamma, a : A^\perp[c], b : B^\perp[c]} \text{ [TYP-RES]}$$

$$\Rightarrow \frac{\text{Lem. 3.10 twice}}{\Omega \vdash P\{a/z, b/w\} @ d''_2 :: \Gamma, a : A^\perp[c], b : B^\perp[c]}$$

By assumption and by definition, $\Omega \Vdash d''_2 \sqsupseteq d'_2$. Also, by definition, $\Omega \Vdash d'_2 \sqsupseteq d'_1 \sqcap d''_2$, so, by assumption, $\Omega \Vdash d'_2 \sqsupseteq d$. Hence, $\Omega \Vdash d''_2 \sqsupseteq d$. ◀

Liveness / Progress. Liveness / Progress properties specify the conditions under which processes can reduce. The progress property of APCP states that reduction takes place for a syntactic notion of “live” processes [30]. Since this result does not rely on APCP’s priority mechanisms, it applies to our process language as well.

4 Logical Relation

This section defines an equivalence on typed processes up to “observable messages” (Def. 4.9) that we will use to state and prove DSNI in Sec. 5. We first give some preliminary definitions in Sec. 4.1, before defining the logical relation that induces this equivalence in Sec. 4.2.

4.1 Preliminary Definitions

As anticipated in Sec. 2.4, we are interested in the behavior of a process when it runs in different contexts. That is, we want to connect all the free names of the process in arbitrary ways. To this end, we define evaluation contexts: processes with a hole inside which a process may reduce (so under parallel composition and restriction). Evaluation contexts are typeset using an orange and monospaced font.

► **Definition 4.1** (Evaluation Context). Evaluation contexts ($\textcolor{orange}{E}$) are defined as follows:

$$\textcolor{orange}{E} ::= [\cdot] \mid \textcolor{orange}{E} \mid P \mid (\nu xy)\textcolor{orange}{E}$$

We write $\textcolor{orange}{E}[P]$ to denote the process obtained by replacing the hole $[\cdot]$ in $\textcolor{orange}{E}$ by P .

Any definitions on processes before and after this definition are lifted to evaluation contexts, without assigning any meaning to the hole. The exception is that alpha renaming does not apply to names that are bound by restriction but not free inside the scope of the restriction.

► **Example 4.2.** The following is an evaluation context:

$$\textcolor{orange}{E} := (\nu uw)(\nu xy)(\nu zv)(x(); u[] \mid z[] \mid [\cdot])$$

Both u and w are bound in $\textcolor{orange}{E}$. Since u appears free within the scope of the restriction as the subject of a close, alpha renaming applies: $\textcolor{orange}{E} \equiv_{\alpha} (\nu aw)(\nu xy)(\nu zv)(x(); a[] \mid z[] \mid [\cdot])$. However, the same does not hold for w : $\textcolor{orange}{E} \not\equiv_{\alpha} (\nu ua)(\nu xy)(\nu zv)(x(); u[] \mid z[] \mid [\cdot])$.

We refer to the names that connect the process and its context as the *interface*. Our logical relation focuses on messages between context and process, i.e., messages that must pass through the interface. The following definition identifies outputs in process and context that are not blocked by prefixes. In particular, $\text{aon}(P)$ is the set of names *along* which P is ready to output, and $\text{acon}(\textcolor{orange}{E})$ is the set of names *to* which the context is ready to output.

► **Definition 4.3** (Active Interface Names). We define the set of active output names of P , denoted $\text{aon}(P)$, as the subjects of non-blocked outputs in P :

$$\begin{array}{lll} \text{aon}(0) := \emptyset & \text{aon}(P \mid Q) := \text{aon}(P) \cup \text{aon}(Q) & \text{aon}((\nu xy)P) := \text{aon}(P) \setminus \{x, y\} \\ \text{aon}(x[]) := \{x\} & \text{aon}(x[a, b]) := \{x\} & \text{aon}(x[b] \triangleleft j) := \{x\} \\ \text{aon}(x(); P) := \emptyset & \text{aon}(x(y, z); P) := \emptyset & \text{aon}(x(z) \triangleright \{i : P_i\}_{i \in I}) := \emptyset \end{array}$$

We define the set of active context output names of $\textcolor{orange}{E}$, denoted $\text{acon}(\textcolor{orange}{E})$, as the names in the interface of $\textcolor{orange}{E}$ that are connected to active output names of $\textcolor{orange}{E}$ through restriction:

$$\text{acon}(\textcolor{orange}{E}) := \{x \mid \exists y, \textcolor{orange}{E}' . (\textcolor{orange}{E} \equiv (\nu xy)\textcolor{orange}{E}' \wedge x \notin \text{fn}(\textcolor{orange}{E}') \wedge y \in \text{aon}(\textcolor{orange}{E}'))\}$$

We define the set of active interface names of E and P as the union of the active context output names of E and the active output names of P :

$$\text{ain}(E, P) := \text{acon}(E) \cup \text{aon}(P)$$

► **Example 4.4.** We illustrate the active interface names between $P := y[] | w(); v(); 0$ and E from Example 4.2. It is easy to see that $\text{aon}(P) = \{y\}$. To determine $\text{acon}(E)$ we search for names in E that are bound by restriction to names used for output, but not used themselves. That is, we look for names in the interface between the context and the containing process, on which the containing process can expect to receive an output from the context. For example, in E , name v is bound to z which is used for an output, while v itself is not used (it appears in the interface). Since there are no further such names, we have $\text{acon}(E) = \{v\}$. As such, $\text{ain}(E, P) = \{y, v\}$.

The interface is where an attacker (cf. Sec. 2.4) may observe the behavior of our process. In this, we assume that the attacker can only observe messages up to a certain secrecy level ξ . As such, our relation is only interested in the behavior of the process on *observable* channels. To this end, we define a projection on typing contexts to filter out unobservable channels. Also, we define when a process in context is well typed with respect to a given typing context of observable channels.

► **Definition 4.5 (Projection and Networks).** Given a secrecy lattice Ω , a secrecy level $\xi \in \text{dom}(\Omega)$, and a typing context Γ , we define the projection $\Gamma \Downarrow_{\Omega} \xi$ as follows:

$$(\Gamma, x : A[c]) \Downarrow_{\Omega} \xi := \begin{cases} (\Gamma \Downarrow_{\Omega} \xi), x : A[c] & \text{if } \Omega \Vdash c \sqsubseteq \xi \\ \Gamma \Downarrow_{\Omega} \xi & \text{if } \Omega \Vdash c \not\sqsubseteq \xi \end{cases} \quad \emptyset \Downarrow_{\Omega} \xi := \emptyset$$

We often omit Ω when it is clear from the context.

We say E and P form a network with interface Γ observable up to ξ under Ω , denoted $(E, P) \in \text{Net}^{\Omega; \xi}(\Gamma)$ if and only if there are d, d', Γ' such that $\Gamma = \Gamma' \Downarrow_{\Omega} \xi$, $\Omega \vdash P @ d' :: \Gamma'$, and $\Omega \vdash E[P] @ d :: \emptyset$. By abuse of notation, we write $(E_1, P_1; E_2, P_2) \in \text{Net}^{\Omega; \xi}(\Gamma)$ to denote $(E_1, P_1) \in \text{Net}^{\Omega; \xi}(\Gamma)$ and $(E_2, P_2) \in \text{Net}^{\Omega; \xi}(\Gamma)$.

► **Example 4.6.** We anticipate illustrating noninterference on the secure running example introduced in Example 3.2 on a *Low* secrecy channel, by considering the projection of its typing context and an evaluation context to form a network. Recall the typing and IFC annotations from Examples 3.7 and 3.8:

$$\vdash \text{Gov}_A^H @ L :: \Gamma' = a_L : \&\{oc_1 : \perp, oc_2 : \perp\}[L], a_I : \oplus\{\text{act} : 1, \text{wait} : 1\}[H].$$

We have $\Gamma := \Gamma' \Downarrow L = a_L : \&\{oc_1 : \perp, oc_2 : \perp\}[L]$. Let $E := (\nu a_H a_L)(\nu a_I i_A)(\text{Gov}_A^L | [\cdot] | \text{Int}_A)$. It is straightforward to confirm that $\vdash E[\text{Gov}_A^H] @ L :: \emptyset$. Hence, $(E, \text{Gov}_A^H) \in \text{Net}^L(\Gamma)$.

In our relation, we want to exhaust reductions on unobservable channels, after which we scrutinize behavior on observable names in the interface. To this end, we define *unobservable reductions*, which entail reductions internal to the process or the context, but also communications between process and context on unobservable interface channels.

► **Definition 4.7 (Unobservable Reduction).** We define unobservable reduction as

$$E, P \longrightarrow_{\Omega; \xi; \Gamma} E', P'$$

if and only if $(E, P) \in \text{Net}^{\Omega; \xi}(\Gamma)$, $E[P] \longrightarrow E'[P']$ and $(E', P') \in \text{Net}^{\Omega; \xi}(\Gamma)$. We write $\longrightarrow_{\Omega; \xi; \Gamma}^?$ (resp. $\longrightarrow_{\Omega; \xi; \Gamma}^*$) for the reflexive (resp. reflexive transitive) closure of $\longrightarrow_{\Omega; \xi; \Gamma}$, and $E, P \not\longrightarrow_{\Omega; \xi; \Gamma}$ to denote that there are no E', P' such that $E, P \longrightarrow_{\Omega; \xi; \Gamma} E', P'$.

$$(E_1[P_1]; E_2[P_2]) \in E^{\Omega; \xi}[\Gamma] \iff \quad (5)$$

$$\wedge (E_1, P_1; E_2, P_2) \in \text{Net}^{\Omega; \xi}(\Gamma) \quad (6)$$

$$\wedge \forall E'_1, P'_1. E_1, P_1 \xrightarrow{*_{\Omega; \xi; \Gamma}} E'_1, P'_1 \xrightarrow{\cdot}_{\Omega; \xi; \Gamma} \quad (7)$$

$$\implies \exists E'_2, P'_2. E_2, P_2 \xrightarrow{*_{\Omega; \xi; \Gamma}} E'_2, P'_2 \xrightarrow{\cdot}_{\Omega; \xi; \Gamma} \quad (8)$$

$$\wedge \forall x \in (\text{ain}(E'_1, P'_1) \cup \text{ain}(E'_2, P'_2)) \cap \text{dom}(\Gamma). (E'_1, P'_1; E'_2, P'_2) \in V_x^{\Omega; \xi}[\Gamma]$$

$$\wedge \text{aon}(P'_1) \cap \text{dom}(\Gamma) = \text{aon}(P'_2) \cap \text{dom}(\Gamma)$$

 **Figure 4** Term interpretation.

Our relation often requires “zooming in” on specific parts of processes. To this end, we define notions to deal with atomic parts of processes.

► **Definition 4.8** (Nodes and Normal Forms). *Given a process P, we say P is a node if P $\not\equiv Q \mid R$, P $\not\equiv (\nu xy)Q$, and P $\not\equiv 0$.*

We say a process $Q = (\nu x_i y_i)_{i \in I} \prod_{j \in J} P_j$ is in normal form if, for every $j \in J$, P_j is a node, and Q is a normal form of P if $P \equiv Q$. Normal forms are closed under structural congruence: every process induces an equivalence class of structurally congruent normal forms.

Given a process in normal form $Q = (\nu x_i y_i)_{i \in I} \prod_{j \in J} P_j$, we define $\text{nodes}(Q) := \{P_j \mid j \in J\}$ and $\text{binders}(Q) := \{\{x_i, y_i\} \mid i \in I\}$. By abuse of notation, given a process P not necessarily in normal form, we write $\text{nodes}(P)$ to denote $\text{nodes}(Q)$ for an arbitrary normal form Q of P.

For example, let

$$P := (\nu uw)((\nu xy)(z(); x[] | y(); u[]) | w(); 0) | 0 \quad Q := (\nu uw)(\nu xy)(z(); x[] | y(); u[] | w(); 0).$$

Then Q is a normal form of P, with $\text{nodes}(Q) = \{z(); x[], y(); u[], w(); 0\}$ and $\text{binders}(Q) = \{\{u, w\}, \{x, y\}\}$.

4.2 The Relation

Having presented all its ingredients, we now introduce our logical relation. As usual, the relation consists of two parts: a *term interpretation* and a *value interpretation*, defined by mutual multiset induction on the interfaces of processes. The term interpretation is the main part of the relation, and is responsible for calling on the value interpretation when a message is ready to be communicated across the observable interface, as well as ensuring deadlock sensitivity of our noninterference result. The value interpretation zooms in on the interface, and ensures that the two runs of the process behave identically on observable messages that are to be communicated across the interface.

Let us start by presenting our term interpretation, denoted $E^{\Omega; \xi}[\Gamma]$, in Fig. 4. It relates pairs of processes, given a secrecy lattice Ω , a secrecy level $\xi \in \text{dom}(\Omega)$, and an interface Γ . We break down its definition part by part. Part (5) implicitly requires each process to be separable into a context E_i and a process P_i . Part (6) then requires the interface between E_i and P_i to correspond to Γ up to observability ξ (cf. Def. 4.5). Part (7) exhausts unobservable reductions for E_1, P_1 (cf. Def. 4.7) in every way possible, resulting in E'_1, P'_1 . Part (8) first requires E_2, P_2 to “catch up” through exhaustive unobservable reductions, resulting in E'_2, P'_2 . Then, Part (8) invokes the value interpretation, presented next, to scrutinize any messages that are ready to be transferred across the observable part of the interface of either E'_i, P'_i (cf. Def. 4.3). Finally, Part (8) ensures deadlock sensitivity by requiring the observable messages

$$\begin{aligned}
& \text{1 } (\mathbf{E}_1, P_1; \mathbf{E}_2, P_2) \in V_x^{\Omega; \xi} [\Gamma, x : 1[c]] \iff ((\mathbf{E}_1, P_1; \mathbf{E}_2, P_2) \in \text{Net}^{\Omega; \xi}(\Gamma, x : 1[c]) \\
& \quad \wedge P_1 \equiv x[] | P'_1 \wedge P_2 \equiv x[] | P'_2 \wedge (\mathbf{E}_1[x[] | P'_1]; \mathbf{E}_2[x[] | P'_2]) \in E^{\Omega; \xi}[\Gamma]) \\
\hline
& \oplus \quad (\mathbf{E}_1, P_1; \mathbf{E}_2, P_2) \in V_x^{\Omega; \xi} [\Gamma, x : \oplus\{i : A_i\}_{i \in I}[c]] \iff & (9) \\
& \quad (\mathbf{E}_1, P_1; \mathbf{E}_2, P_2) \in \text{Net}^{\Omega; \xi}(\Gamma, x : \oplus\{i : A_i\}_{i \in I}[c]) & (10) \\
& \quad \wedge \exists j \in I. x[b_1] \triangleleft j \in \text{nodes}(P_1) \wedge x[b_2] \triangleleft j \in \text{nodes}(P_2) & (11) \\
& \quad \wedge b_1 \in \text{dom}(\Gamma) \implies b_1 = b_2 \wedge P_1 \equiv x[b_1] \triangleleft j | P'_1 \wedge P_2 \equiv x[b_2] \triangleleft j | P'_2 & (12) \\
& \quad \quad \wedge (\mathbf{E}_1[x[b_1] \triangleleft j | P'_1]; \mathbf{E}_2[x[b_2] \triangleleft j | P'_2]) \in E^{\Omega; \xi}[\Gamma \setminus b_1] \\
& \quad \wedge b_1 \notin \text{dom}(\Gamma) \implies (b_2 \notin \text{dom}(\Gamma) & (13) \\
& \quad \quad \wedge P_1 \equiv (\nu b_1 b')(x[b_1] \triangleleft j | P'_1) \wedge P_2 \equiv (\nu b_2 b')(x[b_2] \triangleleft j | P'_2) \\
& \quad \quad \wedge (\mathbf{E}_1[(\nu b_1 b')(x[b_1] \triangleleft j | P'_1)]; \mathbf{E}_2[(\nu b_2 b')(x[b_2] \triangleleft j | P'_2)]) \in E^{\Omega; \xi}[\Gamma, b' : A_j[c]]) \\
\hline
& \otimes \quad (\mathbf{E}_1, P_1; \mathbf{E}_2, P_2) \in V_x^{\Omega; \xi} [\Gamma, x : A \otimes B[c]] \iff \\
& \quad (\mathbf{E}_1, P_1; \mathbf{E}_2, P_2) \in \text{Net}^{\Omega; \xi}(\Gamma, x : A \otimes B[c]) \\
& \quad \wedge x[a_1, b_1] \in \text{nodes}(P_1) \wedge x[a_2, b_2] \in \text{nodes}(P_2) \\
& \quad \wedge a_1, b_1 \in \text{dom}(\Gamma) \implies a_1 = b_1 \wedge a_2 = b_2 \wedge P_1 \equiv x[a_1, b_1] | P'_1 \wedge P_2 \equiv x[a_2, b_2] | P'_2 \\
& \quad \quad \wedge (\mathbf{E}_1[x[a, b] | P'_1]; \mathbf{E}_2[x[a, b] | P'_2]) \in E^{\Omega; \xi}[\Gamma \setminus a, b] \\
& \quad \wedge (a_1 \in \text{dom}(\Gamma) \wedge b_1 \notin \text{dom}(\Gamma)) \implies (a_1 = a_2 \wedge b_2 \notin \text{dom}(\Gamma) \\
& \quad \quad \wedge P_1 \equiv (\nu b_1 b')(x[a_1, b_1] | P'_1) \wedge P_2 \equiv (\nu b_2 b')(x[a_2, b_2] | P'_2) \\
& \quad \quad \wedge (\mathbf{E}_1[(\nu b_1 b')(x[a_1, b_1] | P'_1)]; \mathbf{E}_2[(\nu b_2 b')(x[a_2, b_2] | P'_2)]) \in E^{\Omega; \xi}[\Gamma, b' : B[c] \setminus a]) \\
& \quad \wedge (a_1 \notin \text{dom}(\Gamma) \wedge b_1 \in \text{dom}(\Gamma)) \implies (a_2 \notin \text{dom}(\Gamma) \wedge b_1 = b_2 \\
& \quad \quad \wedge P_1 \equiv (\nu a_1 a')(x[a_1, b_1] | P'_1) \wedge P_2 \equiv (\nu a_2 a')(x[a_2, b_2] | P'_2) \\
& \quad \quad \wedge (\mathbf{E}_1[(\nu a_1 a')(x[a_1, b_1] | P'_1)]; \mathbf{E}_2[(\nu a_2 a')(x[a_2, b_2] | P'_2)]) \in E^{\Omega; \xi}[\Gamma, a' : C[c] \setminus b]) \\
& \quad \wedge a_1, b_1 \notin \text{dom}(\Gamma) \implies (a_2, b_2 \notin \text{dom}(\Gamma) \\
& \quad \quad \wedge P_1 \equiv (\nu a_1 a')(\nu b_1 b')(x[a_1, b_1] | P'_1) \wedge P_2 \equiv (\nu a_2 a')(\nu b_2 b')(x[a_2, b_2] | P'_2) \\
& \quad \quad \wedge (\mathbf{E}_1[(\nu a_1 a')(\nu b_1 b')(x[a_1, b_1] | P'_1)]; \mathbf{E}_2[(\nu a_2 a')(\nu b_2 b')(x[a_2, b_2] | P'_2)]) \in E^{\Omega; \xi}[\Gamma, a' : A[c], b' : B[c]])
\end{aligned}$$

■ **Figure 5** Value interpretation, output cases (1, \oplus , \otimes).

of P'_1 and P'_2 to coincide. In particular, if P'_1 does not produce any observable messages along a name in the interface due to a deadlock imposed by a secret, Part (8) guarantees that P'_2 does not produce any observable messages on that name either.

We present our value interpretation, denoted $V_x^{\Omega; \xi} [\Gamma]$, in Figs. 5 and 6. It relates pairs of context-process tuples $(\mathbf{E}_1, P_1; \mathbf{E}_2, P_2)$, given a secrecy lattice Ω , a secrecy level $\xi \in \text{dom}(\Omega)$, an (observable) interface Γ , and a name $x \in \text{dom}(\Gamma)$. The relation is defined by cases on the type A assigned to x in Γ . If A is output-like (1, \oplus , \otimes ; Fig. 5), the relation looks for a corresponding output on x in the processes to (observably) move across the interface into the contexts²; if A is input-like ($\perp, \&, \otimes$; Fig. 6), the relation looks for a corresponding output on a name y connected by restriction to x in the contexts to (observably) move across the interface into the processes. We detail the representative cases where $A \in \{\oplus, \&\}$.

When $A = \oplus\{i : A_i\}_{i \in I}$ (9), we first check well typedness as usual (10) (cf. Def. 4.5). We then assert that both P_1 and P_2 have ready a selection on x , both on the same label $j \in I$ (11).

² In the rest of the paper, we often write $\Gamma \setminus x$ to denote Γ' given $\Gamma = \Gamma', x : A[c]$.

$$\begin{aligned}
 \perp & \quad (\mathbf{E}_1, P_1; \mathbf{E}_2, P_2) \in V_x^{\Omega; \xi}[\Gamma, x : \perp[c]] \iff ((\mathbf{E}_1, P_1; \mathbf{E}_2, P_2) \in \text{Net}^{\Omega; \xi}(\Gamma, x : \perp[c]) \\
 & \quad \wedge (\mathbf{E}_1 \equiv (\nu yx)(y[] | \mathbf{E}'_1) \wedge \mathbf{E}_2 \equiv (\nu yx)(y[] | \mathbf{E}'_2)) \\
 & \quad \implies (\mathbf{E}'_1[(\nu yx)(y[] | P_1)]; \mathbf{E}'_2[(\nu yx)(y[] | P_2)]) \in E^{\Omega; \xi}[\Gamma]) \\
 \& & \quad (\mathbf{E}_1, P_1; \mathbf{E}_2, P_2) \in V_x^{\Omega; \xi}[\Gamma, x : \&\{i : A_i\}_{i \in [c]}] \iff & (14) \\
 & & \quad (\mathbf{E}_1, P_1; \mathbf{E}_2, P_2) \in \text{Net}^{\Omega; \xi}(\Gamma, x : \&\{i : A_i\}_{i \in [c]}) & (15) \\
 & \wedge & \quad (\exists j \in I. \mathbf{E}_1 \equiv (\nu bb')(\nu yx)(y[b] \triangleleft j | \mathbf{E}'_1) \wedge \mathbf{E}_2 \equiv (\nu bb')(\nu yx)(y[b] \triangleleft j | \mathbf{E}'_2)) & (16) \\
 & \implies & \quad ((\nu bb')\mathbf{E}'_1[(\nu yx)(y[b] \triangleleft j | P_1)]; & (17) \\
 & & \quad (\nu bb')\mathbf{E}'_2[(\nu yx)(y[b] \triangleleft j | P_2)]) \in E^{\Omega; \xi}[\Gamma, b : A_j[c]] \\
 \wp & \quad (\mathbf{E}_1, P_1; \mathbf{E}_2, P_2) \in V_x^{\Omega; \xi}[\Gamma, x : A \wp B[c]] \iff \\
 & & \quad (\mathbf{E}_1, P_1; \mathbf{E}_2, P_2) \in \text{Net}^{\Omega; \xi}(\Gamma, x : A \wp B[c]) \\
 & \wedge & \quad (\mathbf{E}_1 \equiv (\nu aa')(\nu bb')(\nu yx)(y[a, b] | \mathbf{E}'_1) \wedge \mathbf{E}_2 \equiv (\nu aa')(\nu bb')(\nu yx)(y[a, b] | \mathbf{E}'_2)) \\
 & \implies & \quad ((\nu aa')(\nu bb')\mathbf{E}'_1[(\nu yx)(y[a, b] | P_1)]; & \\
 & & \quad (\nu aa')(\nu bb')\mathbf{E}'_2[(\nu yx)(y[a, b] | P_2)]) \in E^{\Omega; \xi}[\Gamma, a : A[c], b : B[c]]
 \end{aligned}$$

■ **Figure 6** Value interpretation, input cases (\perp , $\&$, \wp).

We find that the selections carry continuations b_1 and b_2 , respectively. Since we intend to move the selections across the interface into the contexts, we need to inspect where these b_i are bound: in the context or in the process.

- If b_1 appears in the interface (12), it is bound in \mathbf{E}_1 . We then assert that b_1 and b_2 actually represent the same name, and thus that b_2 is bound in \mathbf{E}_2 . Next, we use structural congruence (Def. 3.3) to separate the selections from the rest of the processes. The case ends with a call on the term interpretation, where the selections have been moved into the contexts. Note that here we remove b_1 ($= b_2$) from the interface, as the processes have relinquished control over this name to their respective contexts: we no longer need to monitor behavior on b_1 . Fig. 7a illustrates this case.
- If b_1 does not appear in the interface (13), it is bound in P_1 . We first assert that b_2 also does not appear in the interface, and thus is bound in P_2 . We then use structural congruence to identify the names to which each b_i is bound – since they are both bound, we conveniently apply alpha conversion and use b' in both cases –, and to separate the selections from the rest of the processes. Finally, we call on the term interpretation, where the selections along with the binders $(\nu b_i b')$ are moved into the contexts. Here, we add b' to the interface, as it must be used in the remainder of the processes, and thus must be monitored. Fig. 7b shows this case.

When $A = \&\{i : A_i\}_{i \in I}$ (14), the processes are expecting a selection from the contexts. We again start with the usual well typedness check (15) (cf. Def. 4.5). The purpose of our relation is to compare runs of the same process in different contexts, and so we cannot make assertions about the readiness of the contexts to make the required selection, or that these are selections of the same label. We therefore proceed only under the condition that indeed the contexts are both ready to select the same label (16). This condition uses structural congruence to identify the names in the contexts to which x is connected, conveniently referred to as y in both contexts. It also identifies the continuations b of the selections and the names b' to which they are connected, as well as the remainder of the contexts. It then calls on the term evaluation (17), where the selections along with the restrictions binding x to y are moved into the processes. As such, x is no longer in the interface, but now the continuations of the selections are: we add b to the interface. Fig. 7c illustrates this case.

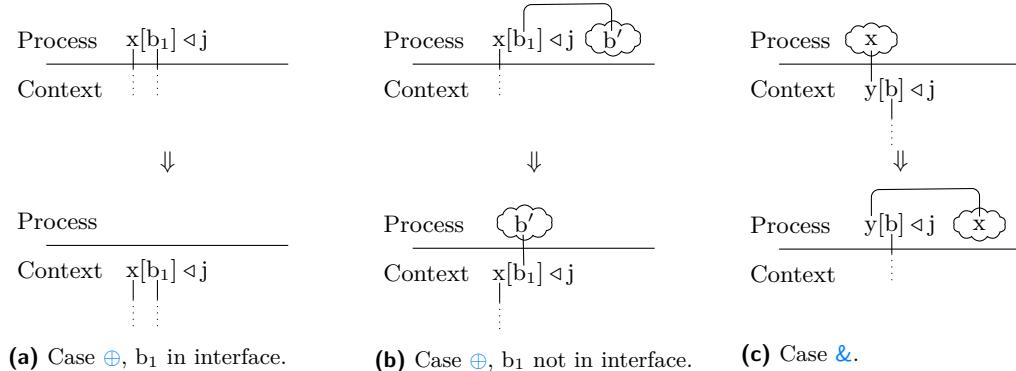


Figure 7 Illustrations of the value interpretation on selections: the selection is moved to/from the process, influencing name connections through the interface. Names in clouds represent parts of the process where the name is used.

Finally, we use our logical relation to define *equivalence up to observable messages*. We say two processes are equivalent up to secrecy level ξ if they agree on their observable interface and they are related by the logical relation when placed inside any two arbitrary evaluation contexts (cf. Def. 4.1). This ensures that, regardless of the context in which the processes run, they will behave the same with respect to the observable interface.

► **Definition 4.9** (Equivalence up to Observable Messages). *The relation*

$$(\Omega \vdash P_1 @ d_1 :: \Gamma_1) \equiv_\xi (\Omega \vdash P_2 @ d_2 :: \Gamma_2)$$

holds if and only if $\Gamma_1 \Downarrow \xi = \Gamma_2 \Downarrow \xi = \Gamma$, and for every E_1, E_2 such that $\Omega \vdash E_1[P_1] @ d_1 :: \emptyset$ and $\Omega \vdash E_2[P_2] @ d_2 :: \emptyset$, $(E_1[P_1]; E_2[P_2]) \in E^{\Omega; \xi}[\Gamma]$ and $(E_2[P_2]; E_1[P_1]) \in E^{\Omega; \xi}[\Gamma]$.

► **Example 4.10.** Consider again the secure variant of Gov_A^H from Example 3.2. Anticipating noninterference, it is straightforward to check that the continuations of the initial branch are equivalent up to observable messages:

$$\begin{aligned} & (\vdash (\nu a_I^{1'} a_I^1)(a_I[a_I^{1'}] \triangleleft \text{act} \mid a_L^1(); a_I^1[])) @ L :: a_I : \oplus\{\text{act} : 1, \text{wait} : 1\}[H], a_L^1 : \perp[L]) \\ & \equiv_L (\vdash (\nu a_I^{1'} a_I^1)(a_I[a_I^{1'}] \triangleleft \text{wait} \mid a_L^1(); a_I^1[])) @ L :: a_I : \oplus\{\text{act} : 1, \text{wait} : 1\}[H], a_L^1 : \perp[L]) \end{aligned}$$

The crucial part is that the different selections on a_I are unobservable.

On the other hand, consider also the insecure variant of Gov_A^H from Example 3.2. Even though their typing contexts are equal (and, hence, so are the projections onto L), the continuations of the initial branch are *not* equivalent up to observable messages:

$$\begin{aligned} & (\vdash (\nu a_L^{1'} a_L^1)(a_L[a_L^{1'}] \triangleleft \text{inf}_1 \mid a_I^1(); a_L^1[])) :: a_L : \oplus\{\text{inf}_1 : 1, \text{inf}_2 : 1\}[L], a_I^1 : \perp[H]) \\ & \not\equiv_L (\vdash (\nu a_L^{1'} a_L^1)(a_L[a_L^{1'}] \triangleleft \text{inf}_2 \mid a_I^1(); a_L^1[])) :: a_L : \oplus\{\text{inf}_1 : 1, \text{inf}_2 : 1\}[L], a_I^1 : \perp[H]) \end{aligned}$$

Here, the different selections on a_L are observable.

5 Deadlock-Sensitive Noninterference (DSNI)

Our main result is that the observable behavior (up to a given secrecy level ξ) of any well-typed process is the same when placed in different contexts. We formalize this using our logical relation (Def. 4.9):

► **Theorem 5.1** (DSNI). *For all secrecy lattices Ω , secrecy levels $\xi \in \text{dom}(\Omega)$ and processes $\Omega \vdash P @ d :: \Gamma$, we have $(\Omega \vdash P @ d :: \Gamma) \equiv_\xi (\Omega \vdash P @ d :: \Gamma)$.*

► **Example 5.2.** Following up on Example 4.10, we can conclude that DSNI holds for the secure variant of Gov_A^H , but not for the insecure variant.

To prove this main result, we prove a more general result (the *fundamental theorem*; Thm. 5.7) that relates two processes through Def. 4.9 given that they are *observably equivalent*. We first define precisely what we mean with observable equivalence before presenting and proving our fundamental theorem in Sec. 5.2.

5.1 Observable Equivalence

Towards defining observable equivalence, we want to identify the nodes (cf. Def. 4.8) of processes that can contribute to messages on observable names in the interface, referred to as *relevant nodes*. Nodes with running secrecy $\not\sqsubseteq \xi$ obviously cannot influence observable interface names. However, nodes with running secrecy $\sqsubseteq \xi$ are not necessarily capable of influencing observable interface names either. In particular, two types of nodes with running secrecy $\sqsubseteq \xi$ cannot influence the observable interface:

- Nodes that input on unobservable names increase their running secrecy after the input, such that they no longer influence observable interface names.
- Nodes that output on unobservable names can only influence nodes that input on unobservable names (and thus cannot influence observable interface names indirectly via the receiving node).

The following notion of *quasi-running secrecy* anticipates these scenarios by assigning a secrecy level to a process based on the influence of its foremost prefix corresponding to the subsequent input/output. It is defined as the join of the current running secrecy of the process and the secrecy level of the name on which the next input/output occurs. If either of the two levels is unobservable, the quasi-running secrecy will be unobservable. In such cases, we know that the foremost prefix of the process cannot influence the observable interface.

► **Definition 5.3** (Quasi-running Secrecy). *Given a node typed $\Omega \vdash P @ d :: \Gamma$, we define the quasi-running secrecy of P , denoted $\text{quasi}(\Omega \vdash P @ d :: \Gamma)$ as follows:*

$$\text{quasi}(\Omega \vdash P @ d :: \Gamma) := \begin{cases} d \sqcup c & \text{if } P = x[] \text{ and } x : 1[c] \in \Gamma \\ d \sqcup c & \text{if } P = x(); P' \text{ and } x : \perp[c] \in \Gamma \\ d \sqcup c & \text{if } P = x[b] \triangleleft j \text{ and } x : \oplus\{i : A_i\}_{i \in I}[c] \in \Gamma \\ d \sqcup c & \text{if } P = x(z) \triangleright \{i : P_i\}_{i \in I} \text{ and } x : \&\{i : A_i\}_{i \in I}[c] \in \Gamma \\ d \sqcup c & \text{if } P = x[a, b] \text{ and } x : A \otimes B[c] \in \Gamma \\ d \sqcup c & \text{if } P = x(y, z); P' \text{ and } x : A \wp B[c] \in \Gamma \end{cases}$$

To compute which nodes of a process are relevant, we start with nodes that have connections to the interface (through free names). We then look at nodes that are connected to these relevant nodes through restrictions. However, not all connections imply a possible influence on the observable interface. Consider a node $x[a, b]$ that is connected to a relevant node on a : the node does not define behavior on a but merely outputs the name, and so a cannot influence the observable interface through this name. For example, in $(\nu xy)(\nu au)(x[a, b] \mid y(z, w); z(); \dots \mid u[])$, the name a is not used for communication until it has been received on y ; hence, the close on u is not considered relevant even if the send on x were relevant. We make this precise by defining *free communication names*: free names that are used as the subjects of unblocked prefixes.

► **Definition 5.4** (Free Communication Names). We define the free communication names of P , denoted $\text{fcn}(P)$, as follows:

$$\begin{array}{ll} \text{fcn}(0) := \emptyset & \\ \text{fcn}(P \mid Q) := \text{fcn}(P) \cup \text{fcn}(Q) & \text{fcn}((\nu xy)P) := \text{fcn}(P) \setminus \{x, y\} \\ \text{fcn}(x[]) := \{x\} & \text{fcn}(x(); P) := \{x\} \cup \text{fcn}(P) \\ \text{fcn}(x[a, b]) := \{x\} & \text{fcn}(x(y, z); P) := \{x\} \cup \text{fcn}(P) \setminus \{y, z\} \\ \text{fcn}(x[b] \triangleleft j) := \{x\} & \text{fcn}(x(z) \triangleright \{i : P_i\}_{i \in I}) := \{x\} \cup \bigcup_{i \in I} \text{fcn}(P_i) \setminus \{z\} \end{array}$$

We now have all the ingredients to determine the relevant nodes of a process. We define the set of relevant nodes of a process inductively by following chains of nodes connected through restriction (of which there are finitely many). We start with nodes connected to the interface directly, and add them if their quasi-running secrecy is $\sqsubseteq \xi$. We then keep adding nodes that are connected to already relevant nodes on observable channels (names with secrecy level $\sqsubseteq \xi$) with quasi-running secrecy $\sqsubseteq \xi$.

► **Definition 5.5** (Relevant Nodes and Binders, and Relevant Form). Suppose given a process in normal form P typed $\Omega \vdash P @ d :: \Gamma$. Suppose every node $Q \in \text{nodes}(P)$ is typed $\Omega \vdash Q @ d_Q :: \Gamma_Q$. Given a secrecy level $\xi \in \text{dom}(\Omega)$, we define the set of relevant nodes of P , denoted $N(P)$, by induction on the size of binders(P) as follows ($N(P) := N_{|\text{binders}(P)|}(P)$):

$$\begin{aligned} N_0(P) &:= \{Q \in \text{nodes}(P) \mid \exists z \in \text{fcn}(Q). z \in \text{dom}(\Gamma \Downarrow \xi) \wedge \text{quasi}(\Omega \vdash Q @ d_Q :: \Gamma_Q) \sqsubseteq \xi\} \\ N_{n+1}(P) &:= N_n(P) \cup \left\{ Q \in \text{nodes}(P) \mid \begin{array}{l} \exists z \in \text{fcn}(Q). \exists Q' \in N_n(P). \exists w : A_w[c] \in \Gamma_{Q'} \\ (\Omega \Vdash c \sqsubseteq \xi \wedge \{z, w\} \in \text{binders}(P)) \\ \wedge \text{quasi}(\Omega \vdash Q @ d_Q :: \Gamma_Q) \sqsubseteq \xi \end{array} \right\} \\ &\forall 0 \leq n < |\text{binders}(P)| \end{aligned}$$

We also define the set of relevant binders of P , denoted $B(P)$, as the subset of $\text{binders}(P)$ used in the inductive step of the definition of $N(P)$. We then define the relevant form of a process in normal form P , denoted $P \Downarrow \xi$, as $(\nu xy)_{\{x, y\} \in B(P)} \prod_{Q \in N(P)} Q$.

Processes are then observably equivalent if their relevant nodes and relevant binders are indistinguishable (up to structural congruence).

► **Definition 5.6** (Observable Equivalence). We say that two processes P and P' are observably equivalent, denoted $P \equiv_\xi P'$, if and only if there are normal forms Q, Q' of P, P' respectively such that $Q \Downarrow \xi \equiv Q' \Downarrow \xi$.

5.2 The Fundamental Theorem

We now state and prove our fundamental theorem, from which DSNI (Thm. 5.1) follows.

► **Theorem 5.7** (Fundamental Theorem). For all secrecy lattices Ω , secrecy levels $\xi \in \text{dom}(\Omega)$ and processes $\Omega \vdash P_1 @ d_1 :: \Gamma_1$ and $\Omega \vdash P_2 @ d_2 :: \Gamma_2$ with $P_1 \equiv_\xi P_2$ and $\Gamma_1 \Downarrow \xi = \Gamma_2 \Downarrow \xi$, we have $(\Omega \vdash P_1 @ d_1 :: \Gamma_1) \equiv_\xi (\Omega \vdash P_2 @ d_2 :: \Gamma_2)$.

We first give several auxiliary results and definitions, before proving Thm. 5.7 on Page 23:

- Lem. 5.8 splits a process that reduces into an evaluation context (Def. 4.1) containing the source of the reduction originating from one of the reduction axioms in Fig. 2 (bottom).
- Lem. 5.9 splits unobservable reduction (Def. 4.7) into one of three cases: reduction internal in the context, reduction internal in the process, and communication between context and process on unobservable names.

- Lem. 5.10 asserts that two observably equivalent (Def. 5.6) processes can “catch up” on each other’s unobservable reductions (Def. 4.7). That is, if one process reduces unobservably, then the other process can do zero or one unobservable reductions such that the resulting processes are again observably equivalent.
- Def. 5.11 defines a *weight* on types and typing context, which we use for induction in the proof of Thm. 5.7 on Page 23.

Lems. 5.8 and 5.9 are proven in the extended paper.

► **Lemma 5.8.** Suppose given a process typed $\Omega \vdash P @ d :: \Gamma$. If $P \rightarrow P'$, then there exists an E for which either of the following holds:

1. $P \equiv E[(\nu xy)(x[] | y(); Q)]$ and $P' \equiv E[Q]$;
2. $P \equiv E[(\nu xy)(x[a, b] | y(z, w); Q)]$ and $P' \equiv E[Q\{a/z, b/w\}]$;
3. $P \equiv E[(\nu xy)(x[b] \triangleleft j | y(w) \triangleright \{i : Q_i\}_{i \in I})]$ for $j \in I$ and $P' \equiv E[Q_j\{b/w\}]$.

► **Lemma 5.9.** Suppose $(E, P) \in \text{Net}^{\Omega; \xi}(\Gamma)$ and $E, P \rightarrow_{\Omega; \xi; \Gamma} E', P'$. Then $E \rightarrow E'$ and $P = P'$, or $P \rightarrow P'$ and $E = E'$, or Lem. 5.8 applies, on names not in Γ .

► **Lemma 5.10 (Catch Up).** Suppose $(E_1, P_1; E_2, P_2) \in \text{Net}^{\Omega; \xi}(\Gamma)$ such that $P_1 \equiv_{\xi} P_2$. If $E_1, P_1 \rightarrow_{\Omega; \xi; \Gamma} E'_1, P'_1$, then there exists P'_2 such that $E_2, P_2 \rightarrow_{\Omega; \xi; \Gamma}^? E_2, P'_2$ and $P'_1 \equiv_{\xi} P'_2$.

Proof. For a smoother proof, we consider a normal form Q_1 of P_1 , and obtain from Q_1 a normal form Q_2 of P_2 such that $Q_1 \Downarrow \xi = Q_2 \Downarrow \xi$. By Def. 4.8, the thesis follows by proving the thesis for these Q_1, Q_2 .

By Lem. 5.9, we can distinguish three cases from which $E_1, Q_1 \rightarrow_{\Omega; \xi; \Gamma} E'_1, Q'_1$ follows.

- (**Internal in context:** $E_1 \rightarrow E'_1$ and $Q_1 = Q'_1$) The thesis holds directly with $Q'_2 := Q_2$.
- (**Internal in process:** $Q_1 \rightarrow Q'_1$ and $E_1 = E'_1$) By Lem. 5.8, Q_1 ’s reduction is due to one of three possible synchronizations inside some evaluation context. Note that Lem. 5.8 may give us processes that are alpha variant to Q_1 and Q'_1 ; in the following we implicitly apply further alpha renaming to match the names in Q_1 and Q'_1 . For space considerations, we sketch only the (**Close-Wait**) case; the other two cases are analogous. Full details are in the extended paper.

We have $Q_1 \equiv F_1[(\nu xy)(x[] | y(); R)] \rightarrow F_1[R] \equiv Q'_1$. The analysis depends on whether the close on x is a relevant node of Q_1 or not.

If not, we derive that the wait on y is also not relevant. It follows by well typedness that the continuation R will neither add relevant nodes nor influence relevancy of other nodes, so $Q_1 \Downarrow \xi = Q'_1 \Downarrow \xi$ and the thesis follows with $Q'_2 := Q_2$.

If the close is indeed a relevant node of Q_1 , we derive that the wait on y and the binder between x and y are also relevant. By assumption, they are then also relevant in Q_2 , so we can derive a similar reduction to Q'_2 .

It remains to show that $Q'_1 \Downarrow \xi \equiv Q'_2 \Downarrow \xi$, which boils down to showing that these processes have coinciding sets of relevant nodes and binders. Both directions of these set inclusions are analogous, so we focus on one: from Q'_1 to Q'_2 . The analysis is by induction on the construction of the sets of relevant nodes and binders. In each case, we consider the appearance of the node: in R or in F_1 . In both cases, a thorough analysis of how the node was included as a relevant node – through a path of relevant binders and nodes that were added before – reveals an analogous relevant node in Q'_2 .

- (**Communication between context and process on names not in Γ**) By definition, the secrecy levels of the involved names are incomparable to ξ . Therefore, none of the nodes involved are relevant or influence relevancy of any other nodes, so $Q_1 \Downarrow \xi = Q'_1 \Downarrow \xi$ and the thesis holds with $Q'_2 := Q_2$. ◀

► **Definition 5.11** (Weight). *The weight of a type A , denoted $\varpi(A)$, is defined as follows:*

$$\begin{aligned}\varpi(1) &:= 1 & \varpi(A \otimes B) &:= \varpi(A) + \varpi(B) + 1 & \varpi(\oplus\{i : A_i\}_{i \in I}) &:= \max_{i \in I}(\varpi(A_i)) + 1 \\ \varpi(\perp) &:= 1 & \varpi(A \wp B) &:= \varpi(A) + \varpi(B) + 1 & \varpi(\&\{i : A_i\}_{i \in I}) &:= \max_{i \in I}(\varpi(A_i)) + 1\end{aligned}$$

The weight of a typing context $\varpi(\Gamma)$ is the sum of the weights of its types.

► **Theorem 5.7** (Fundamental Theorem). *For all secrecy lattices Ω , secrecy levels $\xi \in \text{dom}(\Omega)$ and processes $\Omega \vdash P_1 @ d_1 :: \Gamma_1$ and $\Omega \vdash P_2 @ d_2 :: \Gamma_2$ with $P_1 \equiv_\xi P_2$ and $\Gamma_1 \Downarrow \xi = \Gamma_2 \Downarrow \xi$, we have $(\Omega \vdash P_1 @ d_1 :: \Gamma_1) \equiv_\xi (\Omega \vdash P_2 @ d_2 :: \Gamma_2)$.*

Proof. Let $\Gamma := \Gamma_1 \Downarrow \xi = \Gamma_2 \Downarrow \xi$. Take any E_1, E_2 such that $\Omega \vdash E_1[P_1] @ d'_1 :: \emptyset$ and $\Omega \vdash E_2[P_2] @ d'_2 :: \emptyset$. We need to show that $(E_1[P_1]; E_2[P_2]) \in E^{\Omega; \xi}[\Gamma]$, which we do by induction on $\varpi(\Gamma)$.

The first condition is that $(E_1, P_1; E_2, P_2) \in \text{Net}^{\Omega; \xi}(\Gamma)$; this holds by assumption.

Next, take any E'_1, P'_1 such that $E_1, P_1 \xrightarrow{*_{\Omega; \xi; \Gamma}} E'_1, P'_1 \not\rightarrow_{\Omega; \xi; \Gamma}$. A straightforward induction on the length of these unobservable reductions shows that, by Def. 4.7 and Lem. 5.10, there are E'_2, P'_2 such that $E_2, P_2 \xrightarrow{*_{\Omega; \xi; \Gamma}} E'_2, P'_2 \not\rightarrow_{\Omega; \xi; \Gamma}$, $(E'_1, P'_1; E'_2, P'_2) \in \text{Net}^{\Omega; \xi}(\Gamma)$, and $P'_1 \equiv_\xi P'_2$.

Now, we need to show that, for every $x \in (\text{ain}(E'_1, P'_1) \cup \text{ain}(E'_2, P'_2)) \cap \text{dom}(\Gamma)$,

$$(E'_1, P'_1; E'_2, P'_2) \in V_x^{\Omega; \xi}[\Gamma].$$

Take any such x . Either $x \in \text{ain}(E'_1, P'_1)$ or $x \in \text{ain}(E'_2, P'_2)$; w.l.o.g., assume the former. The rest of the analysis depends on the type of x in Γ .

First, we discuss the output-like cases $(1, \oplus, \otimes)$. In each case, by well typedness, x is the subject of an output-like prefix in P'_1 . Since $x \in \text{ain}(E'_1, P'_1)$, this prefix is unguarded. Since $x \in \text{dom}(\Gamma) = \text{dom}(\Gamma_1 \Downarrow \xi)$, the node in which the prefix appears is relevant in P'_1 . Therefore, since $P'_1 \equiv_\xi P'_2$, there is also a relevant node in P'_2 where this prefix appears unguarded.

For space considerations, we only detail the case where x has type $\oplus\{i : A_i\}[c]$; the other cases are discussed in the extended paper. There exists $j \in I$ such that $x[b_1] \triangleleft j \in \text{nodes}(P'_1)$ and $x[b_2] \triangleleft j \in \text{nodes}(P'_1)$. The analysis depends on whether $b_1 \in \text{dom}(\Gamma)$ or not.

- $(b_1 \in \text{dom}(\Gamma))$ By well typedness, $b_1 \in \text{fn}(P'_1) \cap \text{fn}(P'_2)$. Since $P'_1 \equiv_\xi P'_2$, then $b_1 = b_2$. Hence, $P'_1 \equiv x[b_1] \triangleleft j | P''_1$ and $P'_2 \equiv x[b_2] \triangleleft j | P''_2$. Similar to the case above, and since $\varpi(\Gamma \setminus x) < \varpi(\Gamma)$, it follows from the IH that $(E'_1[x[b_1] \triangleleft j | P''_1]; E'_2[x[b_2] \triangleleft j | P''_2]) \in E^{\Omega; \xi}[\Gamma \setminus x]$. This proves that $(E'_1, P'_1; E'_2, P'_2) \in V_x^{\Omega; \xi}[\Gamma]$.
- $(b_1 \notin \text{dom}(\Gamma))$ By well typedness, $P'_1 \equiv (\nu b_1 b')(x[b_1] \triangleleft j | P''_1)$. The selection on x is a relevant node of P'_1 . Since $P'_1 \equiv_\xi P'_2$, it is also a relevant node of P'_2 . Moreover, $b_2 \notin \text{fn}(P'_2)$: otherwise, $b_2 = b_1$, and then $b_1 \in \text{fn}(P'_1)$. Hence, $P'_2 \equiv (\nu b_2 b')(x[b_2] \triangleleft j | P''_2)$. Clearly, $\Omega \vdash P''_1 @ d''_1 :: \Gamma_1 \setminus x, b'$ and $\Omega \vdash P''_2 @ d''_2 :: \Gamma_2 \setminus x, b'$, and $\Omega \vdash E'_1[(\nu b_1 b')(x[b_1] \triangleleft j | P''_1)] @ d''_1 :: \emptyset$ and $\Omega \vdash E'_2[(\nu b_2 b')(x[b_2] \triangleleft j | P''_2)] @ d''_2 :: \emptyset$. Again, since $P'_1 \equiv_\xi P'_2$, the chain of nodes and binders that are relevant in P'_1 through the binder $(\nu b_1 b')$ has an equivalent such chain in P'_2 through $(\nu b_2 b')$ and the selection on b_2 . Hence, the effect on relevant nodes and binders by removing the binder and the selection on x is the same on P''_1 as it is on P''_2 : $P''_1 \equiv_\xi P''_2$. Clearly, $\Gamma_1 \setminus x, b' \Downarrow \xi = \Gamma_2 \setminus x, b' = \Gamma \setminus x, b'$. Also, $\varpi(A_j) < \varpi(\oplus\{A_i\}_{i \in I})$, so $\varpi(\Gamma \setminus x, b') < \varpi(\Gamma)$. It then follows from the IH that $(E'_1[(\nu b_1 b')(x[b_1] \triangleleft j | P''_1)]; E'_2[(\nu b_2 b')(x[b_2] \triangleleft j | P''_2)]) \in E^{\Omega; \xi}[\Gamma \setminus x, b']$. This proves that $(E'_1, P'_1; E'_2, P'_2) \in V_x^{\Omega; \xi}[\Gamma]$.

Next, we discuss the negative cases (\perp , $\&$, \wp). In each case, by well typedness, x is the subject of an input-like prefix in P'_1 . The context E'_1 binds x to some y by restriction, and E'_1 contains a complementary output-like prefix on y . Following similar reasoning, the same holds for E'_2 . Since $x \in \text{ain}(E'_1, P'_1)$, this output-like prefix appears unguarded in E'_1 . To prove the thesis, we assume that this prefix also appears unguarded in E'_2 .

For space considerations, we only detail the case where x has type $\perp[c]$; the other cases require additional care in handling continuation endpoints and are discussed in the extended paper. We have $E'_1 \equiv (\nu yx)(y[] | E''_1)$ and $E'_2 \equiv (\nu yx)(y[] | E''_2)$. Let $P''_1 := (\nu yx)(y[] | P'_1)$ and $P''_2 := (\nu yx)(y[] | P'_2)$. Clearly, $\Omega \vdash P''_1 @ d''_1 :: \Gamma_1 \setminus x$ and $\Omega \vdash P''_2 @ d''_2 :: \Gamma_2 \setminus x$, and $\Omega \vdash E''_1[P''_1] @ d''_1 :: \emptyset$ and $\Omega \vdash E''_2[P''_2] @ d''_2 :: \emptyset$.

Let Q_1, Q_2 denote the nodes of P'_1, P'_2 , respectively, in which x appears. To prove that $P''_1 \equiv_{\xi} P''_2$, it suffices to show that Q_1 and any related binders are relevant in P''_1 if and only if Q_2 and any related binders are relevant in P''_2 ; any connected nodes/binders follow similar reasoning. We detail only the left-to-right direction; the other direction is analogous. Suppose Q_1 is relevant in P''_1 . Then $\text{quasi}(Q_1) \sqsubseteq \xi$, and thus Q_1 is also relevant in P'_1 through x in the interface. Then also Q_2 is relevant in P'_2 , where $Q_1 \equiv Q_2$ and $\text{quasi}(Q_2) \sqsubseteq \xi$. The analysis depends on how Q_1 is relevant in P''_1 : (i) through the interface, or (ii) through a restriction with another relevant node. In case (i), it follows straightforwardly that Q_2 is also relevant in P'_2 . In case (ii), the connected node is also relevant in P'_1 , and hence there is a related node that is also relevant in P'_2 . Since the two processes agree on observable channels, the channel responsible for including Q_1 as a relevant node of P''_1 is also bound in P''_2 . Then we can conclude that Q_2 is a relevant node of P''_2 .

Since $\varpi(\Gamma \setminus x) < \varpi(\Gamma)$, it then follows from the IH that $(E''_1[P''_1]; E''_2[P''_2]) \in E^{\Omega; \xi}[\Gamma \setminus x]$. This proves that $(E'_1, P'_1; E'_2, P'_2) \in V_x^{\Omega; \xi}[\Gamma]$.

Finally, we show that $\text{aon}(P'_1) \cap \text{dom}(\Gamma) = \text{aon}(P'_2) \cap \text{dom}(\Gamma)$. To prove this set equality, we take any $x \in \text{aon}(P'_1) \cap \text{dom}(\Gamma)$ and prove that $x \in \text{aon}(P'_2) \cap \text{dom}(\Gamma)$; the other direction is analogous. Clearly, x is the subject of an output-like prefix in P'_1 . Since $x \in \text{dom}(\Gamma)$, this output-like prefix must appear unguarded in a node in P'_1 . If the quasi-running secrecy of this node is observable, this node is relevant in P'_1 . Since $P'_1 \equiv_{\xi} P'_2$, P'_2 must also have a relevant node in which the output-like prefix appears unguarded. Otherwise, the node is not relevant in P'_1 , and hence the node in which the output-like prefix appears in P'_2 is also not relevant in P'_2 . Hence, $x \in \text{aon}(P'_2) \cap \text{dom}(\Gamma)$. \blacktriangleleft

DSNI and deadlock freedom. As mentioned in Sec. 3.1, our process language is based on the finite fragment of APCP with priority mechanisms removed. By enriching our process language with APCP’s priority mechanisms, we restrict well typedness to deadlock-free processes. As such, our results remain relevant if we only consider deadlock-free processes.

6 Related Work

Logical relations for session types. Existing logical relations for session types are primarily unary, focusing on proving termination [52, 53, 21]. Binary logical relations have been contributed for proving parametricity [8] and noninterference [20, 5]. All of these logical relations are developed for intuitionistic linear session types, where process networks form trees and, as a result, neither permit cyclic networks nor deadlocks. Whereas our logical relation has its foundations in linear session types, it differs in that it is based on classical linear logic and allows for cycles and deadlocks.

Our work is most similar to prior work by Derakhshan et al. [20] and Balzer et al. [5] on a binary logical relation, in which the authors develop a flow-sensitive IFC type system and use the logical relation to prove noninterference. Our IFC type system is also flow sensitive, but it is designed for an adaptation of APCP (the background of which we discuss separately) that allows for cyclic process networks and deadlocks. Our logical relation resembles the one by the authors in that it employs an interface of names along which observations can be made. In contrast to Derakhshan et al. [20] and Balzer et al. [5], our interface is a set of names with types, rather than a sequent that singles out the providing name from the names being used. The distinction becomes necessary in an intuitionistic linear logic setting, whereas our work is grounded in classical linear logic. The contrast between the intuitionistic and classical setting manifests itself in other aspects of our development, too. For example, in prior intuitionistic IFC session type systems [20, 5], the offering channel always caps the secrecy of the channels in the context and the running secrecy. In our setting, however, the running secrecy of a process and the secrecy levels of its context are not necessarily related. In particular, waiting for a channel to close may increase the running secrecy of a process. Once the channel is closed, it disappears from the context, leaving the running secrecy entirely unrelated to the secracies of the remaining channels.

Like our language, Derakhshan et al. [20]’s language lacks any recursion construct. On the other hand, the possibility of deadlocks introduces a side channel similar to non-termination, which is present in the work by Balzer et al. [5] that supports general recursive session types and thus possibly looping processes. Here, we decided to consider non-recursive processes to focus on side channels through deadlocks only and not through non-termination. In future work we would like to consider scaling our work to support general recursive session types as well. We envision employing an observation index similar to Balzer et al. [5] to stratify the logical relation in the number of observable messages exchanged over the interface. The co-presence of both non-termination and deadlocks will need careful consideration.

Our idea of an observable interface is reminiscent of the free channels with visible communications in prior work by Atkey [4]. Atkey establishes observational equivalence for Wadler’s Classical Processes (CP) by defining a denotational semantics for CP and a logical-relations argument. However, the logical relation in Atkey’s work does not relate two processes of a certain type but rather identifies the possible observations for each type in terms of the input/output behavior of its connectives.

Cyclic process networks. Traditionally, (typed) π -calculi permit cyclic process networks, and as such do not guarantee deadlock freedom. However, since the discovery of Curry-Howard correspondences between linear logic and session types [9, 67], the majority of works on session types restrict their network shapes to trees, such that deadlock freedom is guaranteed. The line of work including APCP (to which our process language is highly related) [42, 51, 19, 29, 30] considers restrictions to session type systems that allow cyclic process networks without deadlocks.

IFC type systems for multiparty session types. Capecchi et al. [13, 11] explore secure information flow with controlled forms of declassification for multiparty sessions and prove a noninterference result via a bisimulation. Our work differs in being flow sensitive and using a binary rather than multiparty session type paradigm. Our use of a logical relation to show noninterference, and our foundations in linear logic also set us apart. Follow-up works by Castellani et al. [14, 12] also study run-time monitoring techniques to ensure secure information flow control in multiparty sessions.

IFC type systems for process calculi. Several approaches have been explored for designing IFC type systems that prevent the leakage of information through message passing in process calculi [34, 35, 15, 16, 17, 36, 18, 26, 25, 41, 68, 55]. Some of these approaches include associating a security label with types or channels [36], associating a security label with actions [35], associating read and write policies with channels [26, 25], and associating a security label with processes and capabilities with expressions [15]. Our approach differs from previous work in having a dynamic running secrecy that makes our system flow sensitive, using session types instead of process calculi, and the design of our novel logical relations for establishing noninterference.

Logical relations for stateful languages. Kripke logical relations have been used to reason about stateful programs [54]. The relation is indexed by a possible world that serves as a semantic model for the heap. It establishes an invariant on the heap and ensures that the invariant is preserved for all future worlds. When combined with step indexing [3, 1], Kripke logical relations can address circularity that arises in higher-order stores [2, 22, 23]. Our logical relation is similar to Kripke logical relations in being developed for a stateful language; names, like locations, are subject to concurrent mutation. However, our session types are rooted in linear logic and thus internalize Kripke’s logical worlds into the type system.

7 Conclusions

We have presented a new session type system with information flow control (IFC) for an asynchronous π -calculus, and a notion of noninterference by means of a logical relation between typed processes. Our development flexibly supports realistic cyclic process networks that may deadlock. As such, our main result is that IFC well typedness implies deadlock-sensitive noninterference (DSNI).

In future work, we plan to study the interplay between IFC / DSNI and several interesting features of message-passing concurrency, such as recursion and non-determinism. As commented on in the previous section, we expect the notion of an observation index [5] to be applicable to circular process networks with recursive processes, although the interplay between potential leakage through non-termination and deadlocks will need careful consideration. Support of non-determinism may be more challenging, especially when combining recursion with non-deterministic choice, which without further restriction permits loop guards at mixed confidentiality level. We will consider focusing on more curtailed but logically motivated notions of non-determinism, such as coexponentials [56], HCP_{ND}^- [43], and linear non-determinism [28]. We are also interested in exploring “name-sensitive” noninterference: the choice of names in outputs is a possible source of information leakage outside the scope of this paper.

References

- 1 Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *15th European Symposium on Programming (ESOP)*, volume 3924 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2006. doi:[10.1007/11693024_6](https://doi.org/10.1007/11693024_6).
- 2 Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. In *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 340–353. ACM, 2009. doi:[10.1145/1480881.1480925](https://doi.org/10.1145/1480881.1480925).
- 3 Andrew W. Appel and David A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(5):657–683, 2001. doi:[10.1145/504709.504712](https://doi.org/10.1145/504709.504712).

- 4 Robert Atkey. Observed communication semantics for classical processes. In *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 26*, pages 56–82. Springer, 2017.
- 5 Stephanie Balzer, Farzaneh Derakhshan, Robert Harper, and Yue Yao. Logical relations for session-typed concurrency. *CoRR*, abs/2309.00192, 2023. doi:10.48550/arXiv.2309.00192.
- 6 Stephanie Balzer and Frank Pfenning. Manifest sharing with session types. *Proceedings of the ACM on Programming Languages*, 1(ICFP):37:1–37:29, 2017. doi:10.1145/3110281.
- 7 Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. Manifest deadlock-freedom for shared session types. In *28th European Symposium on Programming (ESOP)*, volume 11423 of *Lecture Notes in Computer Science*, pages 611–639. Springer, 2019. doi:10.1007/978-3-030-17184-1_22.
- 8 Luís Caires, Jorge A. Pérez, Frank Pfenning, and Bernardo Toninho. Behavioral polymorphism and parametricity in session-based communication. In *22nd European Symposium on Programming (ESOP)*, pages 330–349, 2013. doi:10.1007/978-3-642-37036-6_19.
- 9 Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *21th International Conference on Concurrency Theory (CONCUR)*, volume 6269 of *Lecture Notes in Computer Science*, pages 222–236. Springer, 2010. doi:10.1007/978-3-642-15375-4_16.
- 10 Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Mathematical Structures in Computer Science*, 26(3):367–423, 2016. doi:10.1017/S0960129514000218.
- 11 Sara Capecchi, Ilaria Castellani, and Mariangiola Dezani-Ciancaglini. Typing access control and secure information flow in sessions. *Information and Computation*, 238:68–105, 2014. doi:10.1016/j.ic.2014.07.005.
- 12 Sara Capecchi, Ilaria Castellani, and Mariangiola Dezani-Ciancaglini. Information flow safety in multiparty sessions. *Mathematical Structures in Computer Science*, 26(8):1352–1394, December 2016. doi:10.1017/S0960129514000619.
- 13 Sara Capecchi, Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Tamara Rezk. Session types for access and information flow control. In *21th International Conference on Concurrency Theory (CONCUR)*, pages 237–252, 2010. doi:10.1007/978-3-642-15375-4_17.
- 14 Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Jorge A. Pérez. Self-adaptation and secure information flow in multiparty communications. *Formal Aspects of Computing*, 28(4):669–696, 2016. doi:10.1007/s00165-016-0381-3.
- 15 Silvia Crafa, Michele Bugliesi, and Giuseppe Castagna. Information flow security for boxed ambients. *Electronic Notes in Theoretical Computer Science*, 66(3):76–97, 2002. doi:10.1016/S1571-0661(04)80417-1.
- 16 Silvia Crafa and Sabina Rossi. A theory of noninterference for the π -calculus. In *International Symposium on Trustworthy Global Computing (TGC)*, volume 3705 of *Lecture Notes in Computer Science*, pages 2–18. Springer, 2005. doi:10.1007/11580850_2.
- 17 Silvia Crafa and Sabina Rossi. P-congruences as non-interference for the pi-calculus. In *ACM Workshop on Formal Methods in Security Engineering (FMSE)*, pages 13–22. ACM, 2006. doi:10.1145/1180337.1180339.
- 18 Silvia Crafa and Sabina Rossi. Controlling information release in the π -calculus. *Information and Computation*, 205(8):1235–1273, August 2007. doi:10.1016/j.ic.2007.01.001.
- 19 Ornella Dardha and Simon J. Gay. A New Linear Logic for Deadlock-Free Session-Typed Processes. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Science, pages 91–109. Springer International Publishing, 2018. doi:10.1007/978-3-319-89366-2_5.
- 20 Farzaneh Derakhshan, Stephanie Balzer, and Limin Jia. Session logical relations for noninterference. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–14. IEEE Computer Society, 2021. doi:10.1109/LICS52264.2021.9470654.

- 21 Henry DeYoung, Frank Pfenning, and Klaas Pruiksma. Semi-axiomatic sequent calculus. In *5th International Conference on Formal Structures for Computation and Deduction (FSCD)*, volume 167 of *LIPICS*, pages 29:1–29:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:[10.4230/LIPIcs.FSCD.2020.29](https://doi.org/10.4230/LIPIcs.FSCD.2020.29).
- 22 Derek Dreyer, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. In *15th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 143–156. ACM, 2010. doi:[10.1145/1863543.1863566](https://doi.org/10.1145/1863543.1863566).
- 23 Derek Dreyer, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. *Journal of Functional Programming*, 22(4-5):477–528, 2012. doi:[10.1017/S095679681200024X](https://doi.org/10.1017/S095679681200024X).
- 24 Daniel Hedin and Andrei Sabelfeld. A Perspective on Information-Flow Control. In *Software Safety and Security*, pages 319–347. IOS Press, 2012. doi:[10.3233/978-1-61499-028-4-319](https://doi.org/10.3233/978-1-61499-028-4-319).
- 25 Matthew Hennessy. The security pi-calculus and non-interference. *The Journal of Logic and Algebraic Programming*, 63(1):3–34, April 2005. doi:[10.1016/j.jlap.2004.01.003](https://doi.org/10.1016/j.jlap.2004.01.003).
- 26 Matthew Hennessy and James Riely. Information flow vs. resource access in the asynchronous pi-calculus. *ACM Transactions on Programming Languages and Systems*, 24(5):566–591, September 2002. doi:[10.1145/570886.570890](https://doi.org/10.1145/570886.570890).
- 27 Bas van den Heuvel, Farzaneh Derakhshan, and Stephanie Balzer. Information Flow Control in Cyclic Process Networks, July 2024. doi:[10.48550/arXiv.2407.02304](https://doi.org/10.48550/arXiv.2407.02304).
- 28 Bas van den Heuvel, Joseph W. N. Paulus, Daniele Nantes-Sobrinho, and Jorge A. Pérez. Typed Non-determinism in Functional and Concurrent Calculi. In Chung-Kil Hur, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 112–132, Singapore, 2023. Springer Nature. doi:[10.1007/978-981-99-8311-7_6](https://doi.org/10.1007/978-981-99-8311-7_6).
- 29 Bas van den Heuvel and Jorge A. Pérez. Deadlock freedom for asynchronous and cyclic process networks. In Julien Lange, Anastasia Mavridou, Larisa Safina, and Alceste Scalas, editors, *Proceedings 14th Interaction and Concurrency Experience, Online, 18th June 2021*, volume 347 of *Electronic Proceedings in Theoretical Computer Science*, pages 38–56. Open Publishing Association, 2021. doi:[10.4204/EPTCS.347.3](https://doi.org/10.4204/EPTCS.347.3).
- 30 Bas van den Heuvel and Jorge A. Pérez. A decentralized analysis of multiparty protocols. *Science of Computer Programming*, page 102840, June 2022. doi:[10.1016/j.scico.2022.102840](https://doi.org/10.1016/j.scico.2022.102840).
- 31 C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
- 32 Kohei Honda. Types for dyadic interaction. In *4th International Conference on Concurrency Theory (CONCUR)*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993. doi:[10.1007/3-540-57208-2_35](https://doi.org/10.1007/3-540-57208-2_35).
- 33 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *7th European Symposium on Programming (ESOP)*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. doi:[10.1007/BFb0053567](https://doi.org/10.1007/BFb0053567).
- 34 Kohei Honda, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. Secure information flow as typed process behaviour. In *9th European Symposium on Programming (ESOP)*, volume 1782 of *Lecture Notes in Computer Science*, pages 180–199. Springer, 2000. doi:[10.1007/3-540-46425-5_12](https://doi.org/10.1007/3-540-46425-5_12).
- 35 Kohei Honda and Nobuko Yoshida. A uniform type structure for secure information flow. In *29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 81–92. ACM, 2002. doi:[10.1145/503272.503281](https://doi.org/10.1145/503272.503281).
- 36 Kohei Honda and Nobuko Yoshida. A uniform type structure for secure information flow. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(6):31, 2007. doi:[10.1145/1286821.1286822](https://doi.org/10.1145/1286821.1286822).
- 37 Chung-Kil Hur and Derek Dreyer. A kripke logical relation between ML and assembly. In *38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 133–146. ACM, 2011. doi:[10.1145/1926385.1926402](https://doi.org/10.1145/1926385.1926402).

- 38 Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. In *8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 128–141, 2001. doi:[10.1145/360204.360215](https://doi.org/10.1145/360204.360215).
- 39 Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. *Theoretical Computer Science*, 311(1-3):121–163, 2004. doi:[10.1016/S0304-3975\(03\)00325-6](https://doi.org/10.1016/S0304-3975(03)00325-6).
- 40 Naoki Kobayashi. A partially deadlock-free typed process calculus. In *12th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 128–139. IEEE Computer Society, 1997. doi:[10.1109/LICS.1997.614941](https://doi.org/10.1109/LICS.1997.614941).
- 41 Naoki Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Informatica*, 42(4-5):291–347, 2005. doi:[10.1007/s00236-005-0179-x](https://doi.org/10.1007/s00236-005-0179-x).
- 42 Naoki Kobayashi. A New Type System for Deadlock-Free Processes. In Christel Baier and Holger Hermanns, editors, *CONCUR 2006 – Concurrency Theory*, Lecture Notes in Computer Science, pages 233–247. Springer Berlin Heidelberg, 2006. doi:[10.1007/11817949_16](https://doi.org/10.1007/11817949_16).
- 43 Wen Kokke, J. Garrett Morris, and Philip Wadler. Towards races in linear logic. *Logical Methods in Computer Science*, 16(4), 2020. URL: <https://lmcs.episciences.org/6979>.
- 44 Vasileios Koutavas and Mitchell Wand. Small bisimulations for reasoning about higher-order imperative programs. In *33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 141–152. ACM, 2006. doi:[10.1145/1111037.1111050](https://doi.org/10.1145/1111037.1111050).
- 45 Sam Lindley and J. Garrett Morris. A semantics for propositions as sessions. In *24th European Symposium on Programming (ESOP)*, volume 9032 of *Lecture Notes in Computer Science*, pages 560–584. Springer, 2015. doi:[10.1007/978-3-662-46669-8_23](https://doi.org/10.1007/978-3-662-46669-8_23).
- 46 Sam Lindley and J. Garrett Morris. Talking bananas: Structural recursion for session types. In *21st ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 434–447. ACM, 2016. doi:[10.1145/2951913.2951921](https://doi.org/10.1145/2951913.2951921).
- 47 Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980. doi:[10.1007/3-540-10235-3](https://doi.org/10.1007/3-540-10235-3).
- 48 Robin Milner. *Communication and Concurrency*. PHI Series in Computer Science. Prentice Hall, 1989.
- 49 Robin Milner. *Communicating and Mobile Systems - the Pi-calculus*. Cambridge University Press, 1999.
- 50 Georg Neis, Derek Dreyer, and Andreas Rossberg. Non-parametric parametricity. *Journal of Functional Programming*, 21(4-5):497–562, 2011. doi:[10.1017/S0956796811000165](https://doi.org/10.1017/S0956796811000165).
- 51 Luca Padovani. Deadlock and Lock Freedom in the Linear π -calculus. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS ’14, pages 72:1–72:10, New York, NY, USA, 2014. ACM. doi:[10.1145/2603088.2603116](https://doi.org/10.1145/2603088.2603116).
- 52 Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logical relations for session-based concurrency. In *21st European Symposium on Programming (ESOP)*, volume 7211 of *Lecture Notes in Computer Science*, pages 539–558. Springer, 2012. doi:[10.1007/978-3-642-28869-2_27](https://doi.org/10.1007/978-3-642-28869-2_27).
- 53 Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logical relations and observational equivalences for session-based concurrency. *Information and Computation*, 239:254–302, 2014. doi:[10.1016/j.ic.2014.08.001](https://doi.org/10.1016/j.ic.2014.08.001).
- 54 Andrew M. Pitts and Ian Stark. Operational reasoning for functions with local state. *Higher Order Operational Techniques in Semantics (HOOTS)*, pages 227–273, 1998.
- 55 F. Pottier. A simple view of type-secure information flow in the π -calculus. In *Proceedings 15th IEEE Computer Security Foundations Workshop (CSFW-15)*, pages 320–330, 2002. doi:[10.1109/CSFW.2002.1021826](https://doi.org/10.1109/CSFW.2002.1021826).
- 56 Zesen Qian, G. A. Kavvos, and Lars Birkedal. Client-server sessions in linear logic. *Proceedings of the ACM on Programming Languages*, 5(ICFP):1–31, 2021. doi:[10.1145/3473567](https://doi.org/10.1145/3473567).

- 57 Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal of Selected Areas in Communications*, 21(1):5–19, 2003. doi:10.1109/JSAC.2002.806121.
- 58 Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. Environmental bisimulations for higher-order languages. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 293–302. IEEE Computer Society, 2007. doi:10.1109/LICS.2007.17.
- 59 Davide Sangiorgi and David Walker. *The Pi-Calculus - a Theory of Mobile Processes*. Cambridge University Press, 2001.
- 60 Geoffrey Smith and Dennis M. Volpano. Secure information flow in a multi-threaded imperative language. In *25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 355–364. ACM, 1998. doi:10.1145/268946.268975.
- 61 Kristian Støvring and Søren B. Lassen. A complete, co-inductive syntactic theory of sequential control and state. In *34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 161–172. ACM, 2007. doi:10.1145/1190216.1190244.
- 62 Eijiro Sumii and Benjamin C. Pierce. A bisimulation for type abstraction and recursion. *Journal of the ACM*, 54(5):26, 2007. doi:10.1145/1284320.1284325.
- 63 Jacob Thamsborg and Lars Birkedal. A kripke logical relation for effect-based program transformations. In *16th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 445–456. ACM, 2011. doi:10.1145/2034773.2034831.
- 64 Bernardo Toninho, Luís Caires, and Frank Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *22nd European Symposium on Programming (ESOP)*, volume 7792 of *Lecture Notes in Computer Science*, pages 350–369. Springer, 2013. doi:10.1007/978-3-642-37036-6_20.
- 65 Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996. doi:10.3233/JCS-1996-42-304.
- 66 Philip Wadler. Propositions as sessions. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 273–286. ACM, 2012. doi:10.1145/2364527.2364568.
- 67 Philip Wadler. Propositions as sessions. *Journal of Functional Programming*, 24(2-3):384–418, May 2014. doi:10.1017/S095679681400001X.
- 68 Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *16th IEEE Computer Security Foundations Workshop (CSFW-16 2003), 30 June - 2 July 2003, Pacific Grove, CA, USA*, page 29. IEEE Computer Society, 2003. doi:10.1109/CSFW.2003.1212703.

Refinements for Multiparty Message-Passing Protocols

Specification-Agnostic Theory and Implementation

Martin Vassor 

University of Oxford, UK

Nobuko Yoshida 

University of Oxford, UK

Abstract

Multiparty message-passing protocols are notoriously difficult to design, due to interaction mismatches that lead to errors such as deadlocks. Existing protocol specification formats have been developed to prevent such errors (e.g. multiparty session types (MPST)). In order to further constrain protocols, specifications can be extended with *refinements*, i.e. logical predicates to control the behaviour of the protocol based on previous values exchanged. Unfortunately, existing refinement theories and implementations are tightly coupled with specification formats.

This paper proposes a framework for multiparty message-passing protocols with refinements and its implementation in Rust. Our work *decouples* correctness of refinements from the underlying model of computation, which results in a *specification-agnostic* framework.

Our contributions are threefold. First, we introduce a trace system which characterises *valid refined traces*, i.e. a sequence of sending and receiving actions correct with respect to refinements. Second, we give a correct model of computation named *refined communicating system* (RCS), which is an extension of communicating automata systems with refinements. We prove that RCS only produce valid refined traces. We show how to generate RCS from mainstream protocol specification formats, such as *refined multiparty session types* (RMPST) or *refined choreography automata*. Third, we illustrate the flexibility of the framework by developing both a static analysis technique and an improved model of computation for dynamic refinement evaluation. Finally, we provide a Rust toolchain for decentralised RMPST, evaluate our implementation with a set of benchmarks from the literature, and observe that refinement overhead is negligible.

2012 ACM Subject Classification Software and its engineering → Specification languages; Theory of computation → Assertions; Theory of computation → Concurrency

Keywords and phrases Message-Passing Concurrency, Session Types, Specification

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.41

Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact):*
<https://doi.org/10.4230/DARTS.10.2.23>

Funding Work supported by: EPSRC EP/T00006544/2, EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/2, EP/N028201/1, EP/T014709/2, EP/V000462, EP/X015955/1n NCSS/EPSRC VeTSS, and Horizon EU TaRDIS 101093006.

Acknowledgements We thank B. Ekici, M. Giunti, P. Hou, A. Suresh, and F. Zhou.

1 Introduction

Message passing programming is a notoriously difficult task with new bugs arising with respect to sequential programming, for instance deadlocks. To address this increased complexity, various specifications have been introduced (e.g., message sequence charts [24], multiparty session types [38, 19, 18], choreography automata [1]). In general, specifications are used to



constrain messages, in order to prevent errors such as deadlocks (via message ordering) or payload mismatch (by enforcing the sender and the receiver of a message to agree on the datatype exchanged).

In this paper, we tackle an important and advanced aspect of protocol specification, *logical constraints* (or *contracts*) on asynchronous message-passing communications. Contracts for heterogeneous systems are predominant for correctly designing, implementing, and composing software services, and have a long history in distributed software development as found in Design-by-Contracts [28], Service Level Agreements, and Component-Based Software Engineering. With contracts, software designers can define more precise (refined) and verifiable specifications for distributed software components. Contracts have been investigated from a variety of perspectives, using many different analysis techniques and formalisms. Our goal is to distill an essence of those models for protocol refinements by answering the following questions affirmatively:

- (i) what does it mean for an execution of contracts for message-passing systems to be correct;
- (ii) how do we integrate a theory to a variety of models;
- (iii) how do we analyse their correctness?; and
- (iv) how do we implement correct systems in a programming language?

To explain our framework, consider a *guessing game* (from [41]) with three participants where the first one (participant A) chooses a *secret* integer and sends it to the second participant (B). Then, the third participant (C) tries to *guess* this number. Depending on the guess, B replies with hints (*more* and *less*) until C succeeds in guessing the *correct* value.

The developer writing the specification for such protocol would like to ensure, *in the specification*, that hints from B are consistent with the previous values exchanged. For instance, if the *secret* is 5 and the guess is 10, the specification should constrain B to send *less*. Figure 1 shows a communication diagram of the protocol with constraints (which we call *refinements*) shown in light blue.

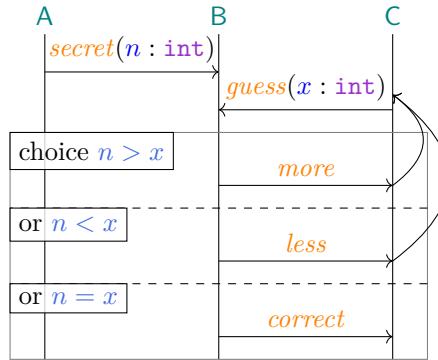


Figure 1 Communication diagram for the guessing game protocol with refinements.

In this paper, we develop a formal framework for refinements, agnostic to any particular specification formalism. Its core part is composed of a characterisation of refinement correctness: *Valid Refined Traces*, and a model of computation: *Refined Communicating Systems* (RCS), where communication is asynchronous and refinements are centrally and dynamically evaluated. For illustration, we use *Refined Multiparty Session Types* as the main specification format for multiparty protocols.

In addition, we demonstrate the versatility of our framework with multiple extensions. First, our framework can accommodate other protocol specification formats (e.g. choreography automata [1]). Second, it is used as a baseline for improved refinement evaluation: we present an optimised model of computation (decentralised refinement evaluation). Finally, it is also used as a baseline for implementing static analysis techniques: we present a simple strategy for statically removing redundant refinements.

Valid Refined Traces. The first building block is a common notion of *correct* executions with respect to added refinements. We introduce *valid refined traces* which are consistent traces with respect to refinements. This approach allows us to establish a general notion of refinements, which is applicable to different logics for constraints, type theories, models of computations, and programming languages. We consider asynchronous communications (FIFOs), distinguishing *sending* and *receiving* actions in traces.

To illustrate our approach, consider the guessing game example shown above. Each execution of that protocol is recorded in a trace, i.e. a sequence of the individual events that take place during the execution (c.f. Section 2.2). For instance, a possible trace of the first four events of the protocol is the following:

$$\text{A!B}\langle\text{secret}, \langle n, 5 \rangle\rangle : \top \cdot \text{A?B}\langle\text{secret}, \langle n, 5 \rangle\rangle : \top \cdot \text{C!B}\langle\text{guess}, \langle x, 5 \rangle\rangle : \top \cdot \text{C?B}\langle\text{guess}, \langle x, 5 \rangle\rangle : \top$$

This trace contains four actions, and each action records an event, i.e. a message emission (denoted with a !) or reception (denoted with a ?). For instance, $\text{A!B}\langle\text{secret}, \langle n, 5 \rangle\rangle : \top$ records **A** sending a message to **B**, the payload of this message is a variable *n*, which has value 5. In the first four actions, we do not need any constraint, therefore actions are guarded by \top which denotes a tautology predicate. The next action following this trace would be for **B** to send either *more*, *correct*, or *less*. Choosing *more* or *less* would be inconsistent with our protocol, since **C** guessed the correct number. For instance, choosing *more* would add the action $\text{B!C}\langle\text{more}\rangle : n > x$ at the end of the queue: the refinement $n > x$ would be violated, since $x = n = 5$.

Valid Refined Traces characterise consistency based on the produced trace; and we aim to provide a model of computation constrained in a way that prevents such inconsistent choices.

Refined Communicating Systems. The second building block of our framework is a model of computation that only produces correct traces. *Communicating Systems* (CS) [5] are a model of concurrent computation, where *Communicating Finite State Machines* communicate asynchronously using unbounded FIFO queues. CS are often used to model and implement MPST [12, 13, 7]. We adapt CS to accommodate refinements, which we call *Refined Communicating Systems* (RCS). The semantics is modified in order to check refinements at every step. For this, we introduce a shared map in order to keep track of variables and their values that are exchanged in messages (e.g. the values of *x* and *n* in the guessing game example). This record of values is used to evaluate refinements, preventing undesired transitions. In this paper, we show that RCS only produce valid refined traces and we explain how to generate an RCS from a RMPST.

Refined MPST. Working with CS is cumbersome, and, in practice, we would prefer to adapt existing specification formats. We present in depth how to integrate refinements in *Multiparty Session Types* (MPST) [38, 19, 18], which are a family of type systems that aims to prevent communication bugs.

The following refined global type (G_{\pm}) is a specification of the guessing game protocol (Figure 1), with refinements: a participant **A** begins by sending a *secret* to **B**; the value of the *secret* is stored in the variable *n*. Then, **C** tries to guess the value (stored in variable *x*),

and \mathbf{B} replies with *more*, *less* (in which case the protocol loops and \mathbf{C} can make another guess: $\mu\mathbf{T}.G$ denotes the recursion) or *correct*, at which point the protocol terminates (*end* denotes the termination). The refinements specify conditions upon which the *more*, *less*, and *correct* branches are possible. For instance, the protocol can take the *correct* branch only if the values in the *secret* and the *guess* messages are the same, i.e. if $x = n$.

$$G_{\pm} = \mathbf{A} \rightarrow \mathbf{B} \left\{ \begin{array}{l} \text{i} \text{ } \mathbf{secret}(n : \text{int} \models \top). \mu\mathbf{T}. \mathbf{C} \rightarrow \mathbf{B} \left\{ \begin{array}{l} \text{i} \text{ } \mathbf{guess}(x : \text{int} \models \top). \mathbf{B} \rightarrow \mathbf{C} \left\{ \begin{array}{l} \text{i} \text{ } \mathbf{more}(\models x < n). \mathbf{T}, \\ \text{i} \text{ } \mathbf{less}(\models x > n). \mathbf{T}, \\ \text{i} \text{ } \mathbf{correct}(\models x = n). \mathbf{end} \end{array} \right\} \end{array} \right\} \end{array} \right\}$$

Compared to standard MPST, *Refined MPST* (RMPST) contain variable names (n and x) and refinements (denoted with $\models r$ in the payloads, meaning that to send the message, r must hold). We present those extensions as well as the relation between RMPST and RCS.

Applications and Extensions. To show the versatility of our framework, we extend it:

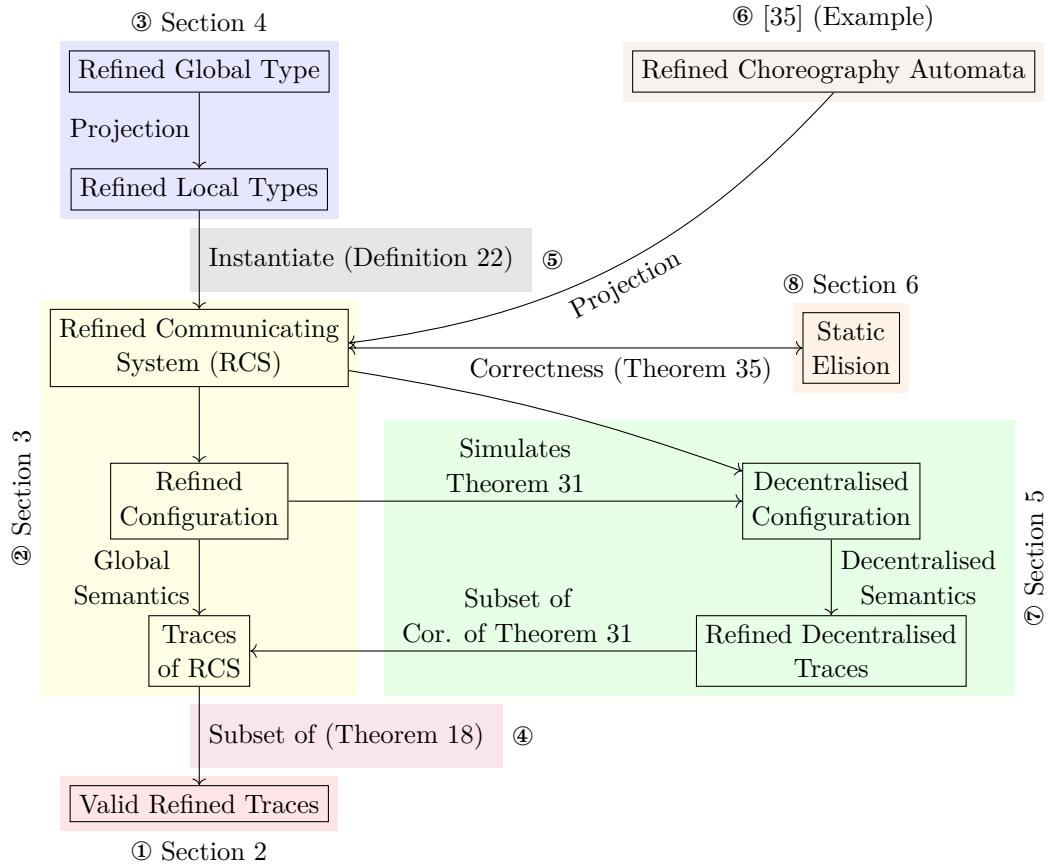
Decentralised Refinement Evaluation: The canonical semantics for RCS we present uses a single shared map of variables to provide a simple way to reason about refinements. Having this global map would not be suited for a distributed implementation. We extend our framework with an alternative semantics where each participant of a protocol has a local map of variables. We show that if variables are not duplicated, then this alternative model also produces valid refined traces.

Static Elision of Redundant Refinements: At places where refinements are redundant (e.g. where it is entailed by previous refinements), we could benefit from removing those refinements. In order to show the versatility of our framework, we show how to develop a simple static analysis technique to remove such redundant refinements.

Refined Choreography Automata: While we mostly use RMPST as an example of protocol specification language, we sketch another specification by (informally) presenting how to integrate refinements in choreography automata (in [35]).

Rust Implementation. The last objective of our work is to implement RMPST into Rust. We choose Rust for several reasons: its affine type system makes it easy to avoid unwanted reuse of values, which helps to prevent a participant from duplicating actions; and thanks to its growing popularity, there are already a few existing toolchains for session types in Rust [27, 6, 26, 25]. Among them, we choose Rumpsteak [7] since it already uses CS to implement MPST participants inside its toolchain. We extend Rumpsteak with refinements using the decentralised refinement evaluation approach. We finally measure the refinement overhead in Rumpsteak.

Contributions and Outline. Our main contribution is to unify the different points presented above in a *single* framework as presented in Figure 2. We introduce a uniform framework which is agnostic to any particular specification formalism, model, semantics and language, defining the correctness of refinements as validity of traces. We then prove the safety of the framework (Theorem 18). We demonstrate the *versatility of our framework* by accommodating *multiple protocol specifications* such as (refined) multiparty session types [38, 19, 18, 42] and (refined) choreography automata [1, 16], *multiple semantics* such as (refined) communicating automata [5] with centralised and decentralised semantics, and *multiple analysis techniques* such as dynamic and static analyses. We provide an implementation of an instance of the framework in Rust. Our framework is the first, to the best of our knowledge, to achieve such versatility.



■ **Figure 2** Overview of the framework for RMPST developed in this work. The coloured backgrounds show the main steps of this paper.

The framework is composed of the following parts (circled numbers refer to Figure 2):

- ① **Valid Refined Traces:** We introduce *valid refined traces* which characterise valid executions with respect to refinements.
- ② **Refined Communicating Systems (RCS):** We extend Communicating Systems to accommodate refinements. From a configuration of RCS, we induce a set of possible traces. One of our main results is Theorem 18 (④), which states that all traces produced by RCS are valid refined traces, which in turn proves the correctness of the RCS.
- ③ **Refined Multiparty Session Types (RMPST):** In Section 4, we adapt MPST (which consists of *global types* (which describe a multiparty protocol), *local types* (which describe the behaviour of a single participant), and a *projection* from global to local types which extracts the behaviour of a single participant) to accommodate for refinements. We show how to generate a RCS from a set of local types with refinements (⑤). In addition, in [35], we sketch how to accommodate refinements in choreography automata, to illustrate the versatility of the framework (⑥).
- ⑦ **and ⑧ Optimisations:** In Section 5 (⑦), we propose a *decentralised* model as an alternative for RCS. We show trace inclusion w.r.t. RCS, which ensures refinements are correctly checked. In Section 7, we implement this improved model in Rust. In addition, in Section 6, we demonstrate how to develop analysis techniques using the framework. We show how redundant refinements can, under some conditions, be statically removed (⑧).

2 Refined Traces and their Validity

This section introduces *refined traces* which are sequences of messages *actions*. We then define their *validity*, introducing two definitions on traces, *well-queued* and *well-predicated* traces. We precede this (in Section 2.1) with preliminary definitions used throughout this paper.

2.1 Preliminaries: Predicates Language and Semantics

This first subsection introduces the basic definitions we use in this paper.

Let \mathbb{V} be a set of variables, ranged over by x, y, \dots ; and a finite set \mathbb{C} of values (in this work, we take 32-bit integers: $\mathbb{Z}/2^{32}\mathbb{Z}$).

We use associative maps from variable names to values, noted M . $\text{dom}(M)$ denotes the domain of a map, that is the set of variables that appear in the map. Maps are equipped with lookup ($M(x)$), update ($M[x \mapsto c]$) and removal ($M \setminus x$) operations. $M_1 \uplus M_2$ denotes the union of M_1 and M_2 if their domains are disjoint (see [35] for the definition of all those operators). Finally, M_\emptyset denotes an empty map.

In order to keep our work general, we do not strictly specify the language of predicates, nor their semantics rules. Instead, we suppose we are given a language to express refinements, whose terms are produced by a rule \mathcal{R} . In this paper, we intentionally leave the logic underspecified so that it can be fine tuned by the end user. In practice, in our implementation (Section 7), custom predicates can easily be added. In the following, we use a simple grammar with arithmetic and relational operators as predicates. Let \mathbb{R} be the set of refinement expressions. We assume refinements can have free variables, and that there exist a function $\text{fv} : \mathbb{R} \rightarrow \mathcal{P}(\mathbb{V})$ that gives the free variables of each refinement expression. We note \mathbb{R}_W be the set of refinements of \mathbb{R} whose set of free variables is $W \subseteq \mathbb{V}$. We assume a variable substitution function, $\mathcal{R}\{v_i/x_i\}$ that substitutes every free occurrence of each variable x_i for the value v_i . For any refinement expression r , $r\{\dots/\text{fv}(r)\}$ is a closed refinement. Since our predicates are abstract, we do not explicitly specify their semantics, nor their well-formedness. Instead, we assume each closed refinement formula evaluates to \top or \perp . We assume there exists a function $\text{eval}(r)$ that evaluates the refinement r , provided that r is closed¹. Finally, we assume the existence of a closed formula \top that is a tautology, i.e. $\text{eval}(\top) = \top$.

Given a map M and a refinement r , we note $M \models r$ if and only if the refinement r is closed under the map M : $\text{fv}(r) \subseteq \text{dom}(M)$, and evaluates to \top after substitution: $\text{eval}(r\{M(\text{fv}(r))/\text{fv}(r)\}) = \top$.

In a protocol with multiple participants, let \mathbb{P} be a set of participants ranged over by A, B, \dots and p, q, \dots being meta-variables over participant names. In this work, messages contain a label, a variable, and a value. Let \mathbb{L} be a set of labels; ℓ and its decorated variants range over labels in \mathbb{L} . We define $\mathbb{M} = \mathbb{L} \times (\mathbb{V} \times \mathbb{C})$ for the set of messages (as a reminder: \mathbb{L} is the set of labels, \mathbb{V} the set of variables, and \mathbb{C} the set of values).

2.2 Traces

Let us denote $\vec{e} = e_1 :: \dots :: e_n$ ($n \geq 0$) as a *FIFO*, i.e., a finite sequence of elements e_i (messages exchanged in this paper). We use ε for an empty FIFO ($n = 0$). We define: $\text{enq}(\vec{e}, e) \stackrel{\text{def}}{=} e :: \vec{e}$; $\text{deq}(\vec{e} :: e) \stackrel{\text{def}}{=} \vec{e}$ ($\text{deq}(\varepsilon)$ is undefined); and $\text{next}(\vec{e} :: e) \stackrel{\text{def}}{=} e$ ($\text{next}(\varepsilon)$ is undefined). Notice

¹ We do not discuss the decidability of the actual chosen logic of refinements here. For undecidable logics, providing such function is, of course, not possible; however this is not in the scope of this work.

that $\text{deq}(\vec{e})$ is defined if and only if $\text{next}(\vec{e})$ is defined. In this paper, we consider one FIFO channel per pair of participant. We call *queues* a map of all pairs of distinct participants to their communication FIFO of a system. We note $\text{enq}_{(p,q)}(w, e)$, $\text{deq}_{(p,q)}(w)$, $\text{next}_{(p,q)}(w)$, where the indices indicates which FIFO of the set is affected (see [35] for the formal definition). We write w_\emptyset for the empty queue, which is the queue where $w_{(p,q)} = \varepsilon$ for all p and q .

Actions are tuples consisting of a sending participant p , a direction of communication $\dagger \in \{!, ?\}$ (! stands for sending, and ? stands for receiving), a receiving participant q , a message m and a predicate r associated to the action (as a reminder: \mathbb{R} is the set of refinements). We require participants to be distinct (i.e. $p \neq q$).

► **Definition 1** (Action and Trace). *An action is an element of \mathbb{A} defined as follows: $\mathbb{A} = \mathbb{P} \times \{!, ?\} \times \mathbb{P} \times \mathbb{M} \times \mathbb{R}$. We write $\alpha = p\dagger q(m) : r$ ($p \neq q$) when $\langle p, \dagger, q, m, r \rangle \in \mathbb{A}$.*

Traces (denoted by τ and its decorated variants) are finite sequences of actions, defined inductively from the rule $\tau ::= \alpha \cdot \tau' \mid \epsilon$, where α is an action. We write \mathbb{A}^ for the set of traces.*

► **Example 2** (Trace). We presented a trace in Section 1.

We denote $\tau_1 \cdot \tau_2$ for the concatenation of two traces. We assume an intuitive notion of the size of trace, as well as lemmas that allow us to infer that, if the size is 0, then the trace is ϵ .

2.3 Properties of Refined Traces

In this subsection, we characterise the *correctness* of traces w.r.t. refinements.

There are two conditions valid traces should verify. First, the sending/reception of messages should be consistent (as with normal MPST). Second, for every action of the trace, predicates that guard the action should hold. We call traces that satisfy message consistency *well-queued traces*, and the traces that satisfy the predicates *well-predicated traces*. In the end, we consider traces that satisfy both conditions: we call those traces *valid refined traces*.

To start with well-queued traces, we first evaluate the impact of a trace on a queue, by looking at the effect of each action on that queue (Definition 3).

► **Definition 3** (Trace Ending Up with Queues, well-queued traces). *A trace τ ends up with the queue w_f w.r.t. a queue w_i if:*

1. *If $\tau = \epsilon$, $w_i = w_f$; and*
 2. *If $\tau = p!q(m) : r \cdot \tau'$, then τ' ends up with w_f w.r.t. $\text{enq}_{(p,q)}(w_i, m)$; and*
 3. *If $\tau = p?q(m) : r \cdot \tau'$, then τ' ends up with w_f w.r.t. $\text{deq}_{(p,q)}(w_i)$ and $\text{next}_{(p,q)}(w_i) = m$.*
- A trace τ is well-queued with regards to the queue w if τ ends up with the empty queue w_\emptyset with respect to an initial queue w .*

A trace τ is valid if τ is well-queued with respect to the empty queue w_\emptyset .

► **Remark 4.** In Definition 3, we say w_i is the *initial* queue.

Regarding well-predicated traces, the idea is to record the latest value of each variable in a map; and to use that map to evaluate refinements (Definition 5).

► **Definition 5** (Well-Predicated Traces). *A trace τ is well-predicated under a map M , if either*

- (i) $\tau = \epsilon$; or
- (ii) $\tau = p\dagger q(l, (x, c)) : r \cdot \tau'$ and $M[x \mapsto c] \models r$ and τ' is well-predicated under $M[x \mapsto c]$.

► **Example 6** (Well-Predicated Traces). In Section 1, we presented the trace τ :

$$\text{A!B}\langle secret, \langle n, 5 \rangle \rangle : \top \cdot \text{A?B}\langle secret, \langle n, 5 \rangle \rangle : \top \cdot \text{C!B}\langle guess, \langle x, 5 \rangle \rangle : \top \cdot \text{C?B}\langle guess, \langle x, 5 \rangle \rangle : \top$$

To illustrate Definition 5, we propose two actions after τ :

- (i) $\tau_1 = \text{B!C}\langle more, \langle _, _ \rangle \rangle : x > n$; and
- (ii) $\tau_2 = \text{B!C}\langle correct, \langle _, _ \rangle \rangle : x = n$.

We can investigate whether $\tau \cdot \tau_1$ (resp. $\tau \cdot \tau_2$) is a well-predicated trace under M_\emptyset . According to Definition 5, we have to investigate whether τ_1 (resp. τ_2) is well predicated under $M = \{\langle n, 5 \rangle, \langle x, 5 \rangle\}$.

For τ_1 , according to Item ii in Definition 5, then $x > n$ must hold under M , which is not the case, therefore $\tau \cdot \tau_1$ is not well-predicated.

Regarding τ_2 , according to Item ii in Definition 5, then $x = n$ must hold under M , which is the case.

Finally, we consider traces that are both valid with respect to predicates and to messages. We call those *Valid Refined Traces*. Our overall goal is to show that our framework only produces such valid refined traces.

► **Definition 7** (Valid Refined Traces). A refined trace τ is valid if

- (i) τ is well-queued with respect to the empty queue w_\emptyset ; and
- (ii) τ is well-predicated under the empty map M_\emptyset .

3 Refined Communicating Automata

In this section, we model message-passing concurrent systems with refinements. We ensure that this model only generates valid refined traces (c.f. Definition 7). Our model of computation is an extension of *communicating systems* (CS) [5, 8], which are sets of Finite State Machines communicating using queues. We introduce *refined communicating systems* (RCS), a variant of CS which accounts for refinements and we show that all traces produced by RCS are valid refined traces (Theorem 18).

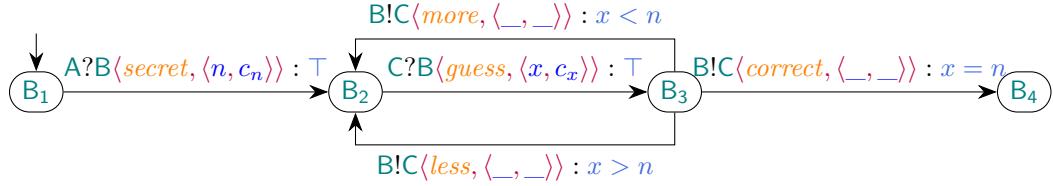
Refined Communicating Finite State Machines. *Communicating systems* [5] are a concurrent model of computation composed of a set of *communicating finite state machines* (CFSM) that interact with exchanges of messages. CFSM are standard finite state machines, where labels represent actions (i.e. sending or receiving messages). Individual FSM are then given a concurrent semantics, which performs messages exchanges. The state of the system is called a *configuration*, which records the state of the individual CFSMs as well as the content of the message queues. In this section, we adapt communicating systems for refinements.

First, we add refinements to the transitions of CFSM, which we call *refined CFSM*. This appears in the additional \mathbb{R} in Definition 8 (we recall \mathbb{R} is the set of refinements).

► **Definition 8** (Refined Communicating Finite State Machine (RCFSM)). An RCFSM is a finite transition system given by $M = \langle Q, C, q_0, \mathbb{M}, \delta \rangle$, where Q is a set of states; $C = \{pq \in \mathbb{P}^2 \mid p \neq q\}$ is a set of channels²; $q_0 \in Q$ is an initial state; \mathbb{M} is a finite alphabet of messages; and $\delta \subseteq Q \times (C \times \{!, ?\} \times \mathbb{A} \times \mathbb{R}) \times Q$ is a finite set of transitions.

We write $s \xrightarrow{\text{itj}(m):r} s'$ for $\langle s, \langle \text{j}, \text{i}, m, r \rangle, s' \rangle \in \delta$. *Refined communicating systems* (RCS) are analogous to their non-refined counterparts and simply consist of a tuple of RCFSM, with one RCFSM per participant. For *refined configurations*, as with (non-refined) configurations,

² The original definition uses *channels*, which we do not use. We keep them for the sake of consistency.



■ **Figure 3** RCFSM of **B** in the G_{\pm} protocol.

we store the states of the individual CFSM and the content of queues. In addition, contrary to non-refined configurations, refined configurations also contain a map in order to keep track of the values of the variables in order to be able to evaluate refinements.

► **Definition 9** (Refined Communicating System (RCS)). *A refined communicating system is a tuple $R = \langle M_p \rangle_{p \in \mathbb{P}}$ of RCFSMs such that $M_p = \langle Q_p, C, q_0 p, \mathbb{M}, \delta_p \rangle$.*

An RCS uses one RCFSM per participant $i \in \mathbb{P}$. A *configuration* represents the state of such RCS, where each participant i is in a local state s_i .

► **Definition 10** (Refined Configuration). *A refined configuration of an RCS R is a tuple S as follows: $S \stackrel{\text{def}}{=} \langle \langle s_1, \dots, s_n \rangle, w, \mathbb{M} \rangle_R$ where each $s_i \in Q_i$, w is a queue of messages, and \mathbb{M} is a map from variables names to values. Let \mathbb{S} be the set of refined configurations.*

► **Remark 11.** Refined configurations are indexed by their RCS. This allows the configuration to store the automaton of the participant. The semantics developed below uses those (local) transitions to infer the global semantics. When the context is clear, we omit this index.

From that, we characterise *initial* and *final* configurations. We call a configuration *initial* when it is a possible configuration where no actions have been taken yet. This means that there is no pending messages (which would imply a previous *send* action), nor known variables (which would imply a previous action initialised the variable). We say a configuration is *final* when there are no pending messages (otherwise, we would expect a *receive* action to take place). Notice that it does not mean the system cannot take action at all.

► **Definition 12** (Initial and Final Refined Configuration). *A refined configuration*

$\langle \langle s_1, \dots, s_n \rangle, w, \mathbb{M} \rangle \in \mathbb{S}$ *is initial if and only if*

- (i) $w = w_{\emptyset}$;
- (ii) $\mathbb{M} = \mathbb{M}_{\emptyset}$; and
- (iii) each s_i is initial in the RCFSM.

A refined configuration $S = \langle \langle s_1, \dots, s_n \rangle, w, \mathbb{M} \rangle \in \mathbb{S}$ is final if and only if $w = w_{\emptyset}$.

► **Example 13** (RCS). The RCFSM of participant **B** in the guessing game is shown in Figure 3. See [35] for the RCFSM of **A** and **C**. Together, they form a RCS, which initial configuration is $\langle \langle A_1, B_1, C_1 \rangle, w_{\emptyset}, \mathbb{M}_{\emptyset} \rangle$.

Refined Semantics. We now define the semantics of RCS in Definition 14 with two reduction rules GRREC and GRSND (the initial GR stands for *global refined*, to distinguish the rules from variants in the following parts of this work), which are respectively used for receiving and sending messages. To avoid confusion with RCFSM reductions, we use a double arrow (\Rightarrow) to represent reductions at the refined communicating system level.

Rule GRSND specifies that, if a participant i reduces from state s_i to state s'_i while sending a message m and if the refinement predicate r attached to the action holds, then the transition is taken at the global level. In the resulting refined configuration, the message is enqueued in the relevant queue and the map of known variables M is updated to take into account the new value of the carried variable c .

Rule GRREC is similar, with the additional requirement that the message received must be the next in the participant's queue (the third premise).

Notice that the verification of refinements is *dynamic*, as it is performed by the corresponding premise in each of the rules, i.e. at execution time.

► **Definition 14** (Refined Global Semantics). *Given a RCS $R = \langle M_p \rangle_{p \in \mathbb{P}}$, we define:*

$$\begin{array}{c} \text{GRREC} \frac{t = s_i \xrightarrow{j?i(\ell, \langle x, c \rangle):r} s'_i \in \delta_i \quad M[x \mapsto c] \models r \quad \text{next}_{(j,i)}(w) = \langle \ell, \langle x, c \rangle \rangle}{\langle \dots, s_i, \dots \rangle, w, M \rangle_R \xrightarrow{t} \langle \dots, s'_i, \dots \rangle, \text{deq}_{(j,i)}(w), M[x \mapsto c] \rangle_R} \\ \\ \text{GRSND} \frac{}{\langle \dots, s_i, \dots \rangle, w, M \rangle_R \xrightarrow{t} \langle \dots, s'_i, \dots \rangle, \text{enq}_{(i,j)}(w, \langle \ell, \langle x, c \rangle \rangle), M[x \mapsto c] \rangle_R} \end{array}$$

► Remark 15. Global transitions are labelled with the underlying local transition. When the local transition is not relevant, we do not show it.

► **Example 16** (Transitions of a RCS). Considering the RCS of G_{\pm} (Figure 3) in its initial configuration $C_i = \langle \langle A_1, B_1, C_1 \rangle, w_{\emptyset}, M_{\emptyset} \rangle$, we have that the automaton of A can fire a transition $A_1 \xrightarrow{A!B(\text{secret}, \langle n, 5 \rangle):T} A_2$, and $M_{\emptyset}[n \mapsto 5] \models T$, by definition of T . Therefore, C_i can take a GRSND transition and reduce to $\langle \langle A_2, B_1, C_1 \rangle, w, \{ \langle n, 5 \rangle \} \rangle$, where w contains a single message $\langle \text{secret}, \langle n, 5 \rangle \rangle$ in $w_{(A,B)}$.

If the RCS is in the configuration $C = \langle \langle A_2, B_3, C_2 \rangle, w_{\emptyset}, M \rangle$ with $M = \{ \langle x, 5 \rangle, \langle n, 5 \rangle \}$, the RCFSM of participant B offers three possible transitions:

- (i) $B_3 \xrightarrow{B!C(\text{more}, \langle _, _ \rangle):x < n} B_2$;
- (ii) $B_3 \xrightarrow{B!C(\text{less}, \langle _, _ \rangle):x > n} B_2$; and
- (iii) $B_3 \xrightarrow{B!C(\text{correct}, \langle _, _ \rangle):x = n} B_4$.

The predicates carried in first two do not hold under M : $M \not\models x < n$ (resp. for $x > n$). Therefore, only $B_3 \xrightarrow{B!C(\text{correct}, \langle _, _ \rangle):x = n} B_4$ is feasible as a GRSND transition in the RCS. As we will see below (Theorem 18), this semantics prevents invalid traces.

Trace of Refined Communicating Systems. In order to show that the semantics of RCS captures the intuition of refinements, we study the traces formed by sequences of reductions (see [35] for the formal definition of traces of RCS).

► **Example 17** (Trace of an RCS). The trace $\tau \cdot \tau_2$ (Example 6) is a trace of the RCS of G_{\pm} .

We conclude this section with our main result, which is that all traces produced by $\mathcal{S}(G)$ are valid refined traces. A trace is *initial* (resp. *final*) if it is obtained from a run whose first (resp. last) state is initial (resp. final).

► **Theorem 18** (Traces of Refined Communicating Systems are Valid Refined Traces). *For all RCS R , for all initial and final traces τ of R , τ is a valid refined trace.*

The proof is in [35].

\mathcal{G}	$\mathbf{p} \rightarrow \mathbf{q}\{\mathbf{l}_i(\mathbf{x}_i : \mathcal{S} \models \mathcal{R}).\mathcal{G}\}_{i \in I} \mid \mu \mathbf{t}.\mathcal{G}$	communication, recursive type
	$ \quad \mathbf{t} \quad \quad \mathbf{end}$	type variable, termination
\mathcal{L}	$\mathbf{p} \oplus \{\mathbf{l}_i(\mathbf{x}_i : \mathcal{S} \models \mathcal{R}).\mathcal{L}\}_{i \in I} \mid \mathbf{t} \mid \mathbf{end}$	internal choice, type variable, termination
	$ \quad \mathbf{p} \& \{\mathbf{l}_i(\mathbf{x}_i : \mathcal{S} \models \mathcal{R}).\mathcal{L}\}_{i \in I} \mid \mu \mathbf{t}.\mathcal{L}$	external choice, recursive type
\mathcal{S}	$\mathbf{int} \mid \dots$	sort (payload types)

■ **Figure 4** Syntax of Global (\mathcal{G}) and Local (\mathcal{L}) Types and Sorts (\mathcal{S}).

4 Refined Multiparty Session Types (RMPST)

In the two previous sections, we introduced refinement validity and a variant of CS which is correct with respect to our validity criterion. However, working with RCS is cumbersome, in particular if we intend to prove additional properties (e.g. deadlock freedom). Fortunately, various models for message-passing concurrent computation have been developed in the literature, many of which can be encoded into CS. Multiparty session types (MPST) [38, 19, 18] is an example of such model. We focus on MPST as they have proved successful for many applications and the theory enjoy many useful properties (e.g. session fidelity, deadlock freedom, liveness etc). However, MPST is not the only possible choice, and we sketch different input models in [35]. In this section, we introduce *refined multiparty session types* (RMPST), which are an extension of MPST annotated with refinement predicates and we show how one can extend existing models to easily obtain refinements.

In Section 4.1, we first present the syntax of *global* and *local* refined multiparty session types, adapted for refinements. In Section 4.2, we present how to obtain RCS from local RMPST, extending a standard approach to implement MPST in CS [12] with refinements.

4.1 Syntax of RMPST

We define the syntax of RMPST. First we assume that messages carry different sorts of payload. As a reminder, for simplicity, in our examples, we only consider `int` payloads. Also, we recall the conventions from Section 2.1: \mathbb{P} is the set of participants and \mathbb{L} is the set of labels. For recursion, we introduce type variables that range over $\{\mathbf{T}, \mathbf{U}, \dots\}$; \mathbf{t} is a meta-variable taken over the set of type variables. We assume all type variables appearing in a type are distinct and we do not (syntactically) distinguish global and local type variables. Finally, \mathbf{x}_i are meta-variables over payload variables taken from the set \mathbb{V} .

We first define *global refined multiparty session types*, which are inductive data types generated by the production \mathcal{G} in Figure 4. The type $\mathbf{A} \rightarrow \mathbf{B}\{\mathbf{l}_i(\mathbf{x}_i : \mathcal{S}_i \models r_i).\mathcal{G}_i\}_{i \in I}$ describes a protocol where \mathbf{A} chooses a label \mathbf{l}_i amongst possible I and sends a message to \mathbf{B} . The message contains a payload of type \mathcal{S}_i , which is bound to \mathbf{x}_i when sent. *Refinement predicates* we introduce guard the communication they are attached to, meaning the system can select a choice with predicate r_i only if r_i holds. In that case, the message is sent and the protocol continues with its continuation of type \mathcal{G}_i . $\mu \mathbf{T}.\mathcal{G}$ binds \mathbf{T} in \mathcal{G} , and a bound type variable \mathbf{T} in a type denotes a protocol recursion. Let $\text{frv}(\mathcal{G})$ denotes the free recursion variables occurring in \mathcal{G} . Finally `end` describes a terminated protocol. Let $\text{parts}(\mathcal{G})$ be the set of participants that appear in \mathcal{G} (c.f. [35] for the definition of $\text{parts}(\mathcal{G})$). We write $\mathbf{p} \in \mathcal{G}$ for $\mathbf{p} \in \text{parts}(\mathcal{G})$.

► **Example 19** (Refined Global Multiparty Session Type). The type \mathcal{G}_\pm presented in Section 1 is a refined global type; we have $\text{parts}(\mathcal{G}_\pm) = \{\mathbf{A}, \mathbf{B}, \mathbf{C}\}$.

To characterise the behaviour of individual participants, we define *refined local multiparty session types*, which are inductive datatypes generated by \mathcal{L} in Figure 4. Recursion, type variables and termination are similar in local and global types. Only the communication specifications differs: in a local type $p \oplus \{l_i(x_i : s_i \models r_i).L_i\}_{i \in I}$ describes an *internal choice*, i.e. the participant chooses a label l_i and sends it to p . Conversely, $p \& \{l_i(x_i : s_i \models r_i).L_i\}_{i \in I}$ describes an *external choice*: p makes a choice amongst the possible l_i and the local participant receives this choice.

Global and local MPST are related: we can *project* a global type onto the local types of its participants. Below, we define a *projection* (partial) operator $G \upharpoonright_p$, which returns the local type of p with respect to the global type G .

We define a projection with a *merge* (partial) operator, which merges multiple local types of a participant into a single local type. This is used to merge the (possibly different) types of the continuations present in the communication branches. The study of different variants of merge operators is an active field (e.g. [32, Section 3]). For the sake of simplicity, in this paper we use a naïve merge operator, which simply ensures that all types are the same.

► **Definition 20** (Projection). *Given p , q and r three distinct participants:*

$$p \rightarrow q \{l_i(x_i : s_i \models R_i).G_i\}_{i \in I} \upharpoonright_p = q \oplus \{l_i(x_i : s_i \models R_i).G_i \upharpoonright_p\}_{i \in I}$$

$$q \rightarrow p \{l_i(x_i : s_i \models R_i).G_i\}_{i \in I} \upharpoonright_p = q \& \{l_i(x_i : s_i \models \top).G_i \upharpoonright_p\}_{i \in I}$$

$$q \rightarrow r \{l_i(x_i : s_i \models R_i).G_i\}_{i \in I} \upharpoonright_p = \sqcap_{i \in I} (G_i \upharpoonright_p)$$

$$\mu t. G' \upharpoonright_p = \begin{cases} \mu t. (G' \upharpoonright_p) & \text{if } p \in G' \text{ or } \text{frv}(G') \neq \emptyset \\ \text{end} & \text{otherwise} \end{cases} \quad t \upharpoonright_p = \text{tend} \upharpoonright_p = \text{end}$$

where a merge operator is defined as: $\sqcap_{i \in I} L_i \stackrel{\text{def}}{=} L$ if $\forall i \in I \cdot L = L_i$, undefined otherwise.

Notice that our local RMPST accept refinements on both receiving and sending, and the semantics developed in Section 3 accept any position for verification. When projecting a global type $G = A \rightarrow B \{l(x : \text{int} \models r).\text{end}\}$ onto local types, we therefore have a choice to project the refinement:

- on the send side: $G \upharpoonright_A = B \oplus \{l(x : \text{int} \models r).\text{end}\}$ and $G \upharpoonright_B = A \& \{l(x : \text{int} \models \top).\text{end}\}$
- on the receive side: $G \upharpoonright_A = B \oplus \{l(x : \text{int} \models \top).\text{end}\}$ and $G \upharpoonright_B = A \& \{l(x : \text{int} \models r).\text{end}\}$
- or a combination of both³.

Our projection takes the first option, i.e. refinements are checked when the message is emitted, but with any of these choices, our developments would not substantially change.

► **Example 21** (Projection). We project G_\pm (Section 1) onto participants A and B ⁴:

$$G_\pm \upharpoonright_A = B \oplus \{\text{secret}(n : \text{int} \models \top).\text{end}\}$$

$$G_\pm \upharpoonright_B =$$

$$A \& \left\{ \text{secret}(n : \text{int} \models \top).\mu T.C \& \left\{ \text{guess}(x : \text{int} \models \top).C \oplus \left\{ \begin{array}{l} \text{more}(\models x < n).T, \\ \text{less}(\models x > n).T, \\ \text{correct}(\models x = n).\text{end} \end{array} \right\} \right\} \right\}$$

³ For instance, if we want to implement a centralised server that communicates with (isolated) clients, we may want all refinements to be asserted by the server, independently of the direction.

⁴ The projection onto C is similar to the recursive part of the projection onto B , with $!$ and $?$ swapped.

4.2 From Refined MPST to Refined Communicating System

In this subsection, we show how to generate an RCS from local RMPSTs. As shown in Definition 20, local types are projected from global multiparty session types. Therefore, this step allows us to complete the conversion from a global RMPST into an RCS. We adapt the translation from local type to CFSMs presented in [13] to accommodate refinements in types.

The intuition behind the translation is to decompose a local type into the individual steps it specifies. For this, we first need to retrieve all those steps. We define the set of types that occur nested in another type: a type T' occurs in a type T (noted $T' \in T$) if it appears in the continuations of T after one or multiple steps (see [35]).

Given this, we can proceed to the translation itself, in Definition 22. This definition says that the states of the RCFSM of a local type T_0 are composed of the (sub)types that appear in T_0 , stripped of the leading μt . (the function `strip` removes the leading recursions variables; this formalises [13, Item (2) in Definition 3.4]) and of recursive variables t . We define the set of transitions of this RCFSM by taking the action each type (i.e. each state) can take, and adding a transition with this action from the initial state to the continuation (stripped of leading μt). In the case that the continuation is a recursion variable t , we have to search in the original type the continuation. Compared to [13, Item (2) in Definition 3.4], we simply add the support for the refinements predicates, which appear both in the types (i.e. in the states) and in the actions (i.e. in the transitions).

► **Definition 22** (RCFSM of Refined Local Types (extends Definition 3.5 in [13])). *Given a global type G , the RCFSM of participant p (with local type $T_0 = G|_p$) is the automaton $\mathcal{A}(T_0) = \langle Q, C, \text{strip}(T_0), \mathbb{M}, \delta \rangle$ where:*

- $Q = \{T' \mid T' \in T_0 \wedge T' \neq t \wedge T' \neq \mu t.T_\mu\}$;
- $C = \{\mathbf{pq} \mid p, q \in G, p \neq q\}$; and
- δ is the smallest set of transitions such that: for all $T \in T_0$ in Q , for all $c \in \mathbb{C}$:
 - if T is $q \oplus \{\ell_i(x_i : S_i \models r_i).T_i\}_{i \in I}$, for all T_i :
 - * if $T_i \neq t$, then $\langle T, p!q(\ell_i, \langle x, c \rangle) : r, \text{strip}(T_i) \rangle \in \delta$
 - * if $T_i = t$ with $\mu t.T' \in T_0$, then $\langle T, p!q(\ell_i, \langle x, c \rangle) : r, \text{strip}(T') \rangle \in \delta$
 - if T is $q \& \{\ell_i(x_i : S_i \models r_i).T_i\}_{i \in I}$, for all T_i :
 - * if $T_i \neq t$, then $\langle T, q?p(\ell_i, \langle x, c \rangle) : r, \text{strip}(T_i) \rangle \in \delta$
 - * if $T_i = t$ with $\mu t.T' \in T_0$, then $\langle T, q?p(\ell_i, \langle x, c \rangle) : r, \text{strip}(T') \rangle \in \delta$

where $\text{strip}(T) \stackrel{\text{def}}{=} \text{strip}(T')$ if $T = \mu t.T'$; and $\text{strip}(T) \stackrel{\text{def}}{=} T$ otherwise.

Finally, we define the *RCS of a type*.

► **Definition 23** (Refined Communicating System of a Type). *The RCS of a type G , noted $\mathcal{S}(G)$, is a tuple composed of the RCFSM of all participants $\mathcal{S}(G) \stackrel{\text{def}}{=} \langle \mathcal{A}(G|_p) \rangle_{p \in \text{parts}(G)}$. We note $\mathcal{C}(G)$ the initial configuration of $\mathcal{S}(G)$.*

► **Example 24** (Refined Communicating System of G_\pm). The communicating system of G_\pm is $\mathcal{S}(G_\pm) = \langle \mathcal{A}(G_\pm|_A), \mathcal{A}(G_\pm|_B), \mathcal{A}(G_\pm|_C) \rangle$.

The initial configuration $\mathcal{C}(G_\pm)$ of this RCS $\mathcal{S}(G_\pm)$ is $\langle \langle A_1, B_1, C_1 \rangle, w_\emptyset, M_\emptyset \rangle$.

Theorem 18 applies to RCS obtained from RMPST: RCS generated from Definition 23 only produce valid refined traces, with the refined global semantics presented in Definition 14. Notice also that, if refinements always hold, RMPST and their semantics coincide with the semantics presented in [12].

5 Decentralised Refined Multiparty Session Types

In the previous section, we presented RCS and we showed that every trace of an RCS is a valid refined trace. However, RCS are theoretical constructions and are not intended to be implemented directly, as they use a global shared map of variables. In practice, a user may want to develop more precise analysis techniques on specific classes of RCS to remove this global map, which allows a decentralised verification of refinements, while keeping the validity of refined traces.

The goal of this section is twofold: on the one hand, the decentralised semantics we develop serves as a theoretical background for our implementation (Section 7). On the other hand, it illustrates the modularity of our framework. We show that the decentralised approach produces valid refined traces by showing refined configurations we developed in Section 3 simulate decentralised systems. This approach is not specific to our variant: we expect other optimisations presented in the literature could be integrated similarly.

This section is divided in the following steps: first, we define what we mean by *decentralised verification of the refinements*, by adapting the semantics of RCS (Definitions 25 and 28). We split the global map of variables' values into local maps (one per participant). Then, we show that despite being modified, the new variant still produces valid refined traces (Definition 7). We justify this claim by proving that under some conditions, the original RCS *simulates* (c.f. [30, Exercise 1.4.17, p. 26]) the decentralised variant (Theorem 31). Since trace equivalence is coarser than simulation, this is sufficient to prove that decentralised configurations that meet the said conditions produce valid refined traces.

The conditions we mentioned above are:

- (i) variables should not be duplicated; and
- (ii) when evaluating a predicate, the free variables of the predicate must be in the local map.

Without the first condition, we can possibly have two distinct values assigned to the same variable without being able to distinguish which is the most recent. The second condition is required to verify the refinements locally (e.g. predicates that constraint an action of A should be checked by A itself). To illustrate the importance of the first condition, consider the type $\text{A} \rightarrow \text{B} \{ \ell_1(x : \text{int}).\text{C} \rightarrow \text{D} \{ \ell_2(x : \text{int}).\text{end} \} \}$. In the centralised approach, x is aliased, while in the decentralised approach, the x exchanged between A and B is stored in a local map, and the x exchanged between C and D is stored in another local map; both are not aliased. To prevent different semantics, we need to prevent such difference, which is the goal of the first condition.

Decentralised Configurations and Decentralised Semantics. First, we define *decentralised* configurations in Definition 25. Compared to Definition 10, instead of a global map in the tuple, we associate a local map to each automata state. Those maps store the variables each participant has access to.

► **Definition 25** (Decentralised Configuration). A Decentralised Configuration of an RCS $\mathcal{S}(\text{G}) = \langle\langle Q_i, C_i, q_{0,i}, \mathbb{A}, \delta_i \rangle\rangle_{i \in \text{parts}(\text{G})}$ is a tuple $\langle\langle \langle s_1, M_1 \rangle, \dots, \langle s_n, M_n \rangle \rangle, w \rangle_{\mathcal{S}(\text{G})}$ where each $s_i \in Q_i$, each M_i is a local map of variables to values, and w is a queue of messages.

Let \mathbb{S}_D be the set of decentralised configurations. We note $\mathcal{D}(\text{G})$ the initial decentralised configuration of $\mathcal{S}(\text{G})$.

Remark 11 also applies for decentralised configurations.

► **Example 26** (Initial decentralised configuration of G_{\pm}). In Example 13, we presented the refined communicating system of G_{\pm} and its associated refined configuration. The *initial decentralised configuration* of this system is $\langle \langle A_1, M_{\emptyset} \rangle, \langle B_1, M_{\emptyset} \rangle, \langle C_1, M_{\emptyset} \rangle, w_{\emptyset} \rangle$. In particular, notice that it uses the same set of refined CFSM than the refined configuration.

The global reduction rules are adapted accordingly: in the rules DREC and DSND (“D” stands for “decentralised”), when a message is sent or received, the corresponding local map is updated, instead of a global map as in GRREC and GRSND.

► **Remark 27.** Contrary to Definition 14, when a variable is sent, it is removed from the local map of variables. Intuitively, when a participant sends a variable, it erases its knowledge of it, to prevent aliasing issues. A direct consequence of this is that, in the centralised implementation, the global map of variables is a *superset* of the local maps in the corresponding decentralised implementation. Indeed, while a variable is in transit, it appears neither in the sender’s map, nor in the receiver’s one. This observation will be proved together with the simulation proof (Theorem 31).

► **Definition 28** (Decentralised Global Semantics). *Given an RCS $R = \langle M_p \rangle_{p \in \mathbb{P}}$*

$$\begin{array}{c} DREC \quad \frac{t = s_i \xrightarrow{j?i(\ell, \langle x, c \rangle):r} s'_i \in \delta_i \quad \text{next}_{(j,i)}(w) = \langle \ell, \langle x, c \rangle \rangle \quad M_i[x \mapsto c] \models r}{\langle \langle \dots, \langle s_i, M_i \rangle, \dots \rangle, w \rangle_R \xrightarrow{t} \langle \langle \dots, \langle s_i, M_i[x \mapsto c] \rangle, \dots \rangle, \text{deq}_{(j,i)}(w) \rangle_R} \\ DSND \quad \frac{}{\langle \langle \dots, \langle s_i, M_i \rangle, \dots \rangle, w \rangle_R \xrightarrow{t} \langle \langle \dots, \langle s_i, M_i \setminus x \rangle, \dots \rangle, \text{enq}_{(i,j)}(w, \langle \ell, \langle x, c \rangle \rangle) \rangle_R} \end{array}$$

Conditions for Decentralised Verification and Correctness Proofs. We now focus on proving that this decentralised semantics produces valid refined traces. As we mentioned above, this holds under two conditions, which we define first:

► **Definition 29** (Conditions for Decentralised Verification Simulation). *Given a decentralised configuration $\langle \langle s_i, M_i \rangle, \dots \rangle, w \rangle$, the conditions for simulation are:*

1. *No duplication:*
 - a. if $\exists M_i \cdot x \in \text{dom}(M_i)$, then $\forall i, j \cdot x \notin w_{(i,j)}$ and $\forall j \neq i \cdot x \notin \text{dom}(M_j)$.
 - b. if $\exists \langle i, j \rangle \cdot x \in w_{(i,j)}$, then $\forall i \cdot x \notin \text{dom}(M_i)$ and $\forall \langle i', j' \rangle \neq \langle i, j \rangle \cdot x \notin w_{(i',j')}$.
2. *Free variables are in the map:* $\forall i \cdot \forall s'_i \cdot s_i \xrightarrow{i?j(\ell, \langle x, c \rangle):r} s'_i \cdot \text{fv}(r) \subseteq \text{dom}(M_i[x \mapsto c])$

► **Definition 30** (Decentralisable Type). *A type G is decentralisable if the two conditions hold for all reachable decentralised configurations from $\mathcal{D}(G)$.*

Notice that the second condition is redundant, as the condition $M_i[x \mapsto c] \models r$ (in the premises of the reduction rules) already requires that $\text{fv}(r)$ is a subset of the variables in $M_i[x \mapsto c]$. Even without making this condition explicit, the system would stall if a predicate cannot be verified. For the sake of clarity, we keep it explicit in the two required conditions.

We now observe a correspondence between the (centralised) refined configuration and the decentralised configuration of a global type G . To characterise the correspondence between centralised and decentralised configuration, we establish a *simulation* relation between the two (see [35] and [30]). Intuitively, a simulation captures the fact that one system (the centralised configuration in our case) can mimic all transitions of another system (the decentralised one here).

We can now prove the main result of this section, which is that the decentralised semantics does not induce new (unwanted) behaviours, i.e. all decentralised transitions can be mimicked by centralised transitions, i.e. the centralised approach simulates the decentralised one.

► **Theorem 31** (Centralised simulates Decentralised). *For all decentralisable RMPST G (Definition 30), $\mathcal{C}(G)$ simulates $\mathcal{D}(G)$.*

Proof. The proof is available in [35]. ◀

This result shows that any type that verifies the conditions stated in Definition 29 can be verified in a decentralised way. The difficulty is that the conditions are about the execution: we do not know whether a predicate will have a missing variable during the execution. With a knowledge flow algorithm, we can infer (from the communication specifications in the global type) which participant has access to which variables at any point in the execution of the protocol, i.e. we can *localise* each variable throughout the execution of the protocol. This algorithm (which we present in [35]) does not present major challenges.

Notice that the reverse simulation does not hold: $\mathcal{D}(G)$ does not simulate $\mathcal{C}(G)$. Indeed, $\mathcal{C}(G)$ can verify a predicate whose variables are spread over different participants, i.e. where variables would be spread across multiple M_i in the decentralised variant.

6 Static Elision of Redundant Refinements

In this section, we present a second optimisation, which is complimentary from the first one. The main idea is to statically analyse a given protocol to find and remove redundant refinements. Our approach is to consider a *target* transition, which we aim to remove the refinement, if possible. Our optimisation can then be applied successively to different target transitions one after each other. For instance, consider the following protocol G_s . We target the second refinement, $x < 10$, which necessarily holds if the first one does (since x does not change). Therefore it is redundant and can be removed.

$$G_s = A \rightarrow B \{\ell_1(x : \text{int} \models x < 0).A \rightarrow B \{\ell_2(y : \text{int} \models x < 10).\text{end}\}\}$$

However, removing refinements is not always trivial, since the communication semantics is asynchronous. Consider for instance the following type:

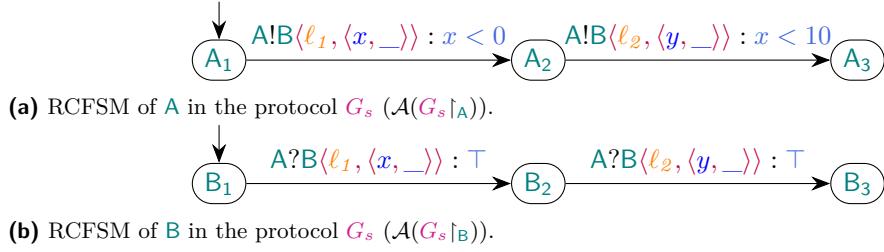
$$A \rightarrow C \{\ell_1(x : \text{int} \models x > 20).A \rightarrow B \{\ell_2(x : \text{int} \models x < 0).C \rightarrow B \{\ell_3(y : \text{int} \models x < 10).\text{end}\}\}\}$$

A naïve approach would be to remove the refinement of the last communication ($x < 10$), since the previous communication has a stronger guarantee ($x < 0$). However, due to the asynchrony of communications, the second and third communications could be swapped at runtime, but the refinement ($x < 10$) does not hold before the action $A \rightarrow B \{\ell_2(x : \text{int} \models x < 0)\dots\}$ occurs. Therefore, in this case, removing the last refinement is incorrect. The optimisation we present takes into account those cases, by keeping track of causal relations between actions.

This section is independent of the previous one, although this second optimisation can help to make some protocols localisable: for instance, G_s above is not localisable. Since the second step $A \rightarrow B \{\ell_2(y : \text{int} \models x < 10).\text{end}\}$ requires A to access x , which is at B . However, once removed, the protocol becomes localisable, and can therefore be decentralised, helping the first optimisation introduced in Section 5.

As with the previous section, the optimisation we present could easily be further improved. Here, we focus on a simple case, as our goal is not to discuss the optimisation itself, but rather to show the versatility of the framework.

We present this section in two steps: first, in Section 6.1, we focus on RCS, which form the core of our framework; then, in Section 6.2, we apply the above result to RMPST.



■ **Figure 5** RCFSM of the RCS of G_s , the running example of Section 6.

6.1 Static Elision of Refinements in RCS

In a first step, we develop and prove the correctness of our analysis in RCS. The question is therefore whether, given a RCS R with one CFSM containing a transition with refinement r , this RCS R is equivalent (bisimilar) to an RCS where r is replaced with T .

For the sake of simplicity, in this subsection, we'll explain the static elision of refinements in RCS using examples. Formal definitions, lemmas and their proofs are available in [35]. We use the RCS of G_s shown in Figure 5.

If we aim to i.e. transitions which payload modify variables that do not appear free in the refinement of the considered transition.

► **Example 32** (Independent transitions). In $S(G_s)$, $A_2 \xrightarrow{A!B<ell_2, <y, __> : x < 10} A_3$ depends on the variable $x \in \text{fv}(x < 10)$. This transition is self-independent. Since the payload of $A_1 \xrightarrow{A!B<ell_1, <x, __> : x < 0} A_2$ is x , the former transition depends on the later.

► **Remark 33.** We note \mathbb{T}_x the set of transitions $\sigma \xrightarrow{\dagger \langle __, <x, __> \rangle} \sigma'$. Given a transition t with refinement r , if $x \in \text{fv}(r)$, then t depends on all transitions of \mathbb{T}_x .

Essentially, when attempting to remove a refinement from a target transition t , we can disregard all transitions t is independent of.

The second definition we will need is about transitions being *well-defined*. So far, nothing prevents us to use refinements with undefined free variables, we simply consider the refinement does not hold (c.f. Definition 14). In this section, we specifically focus on systems where free variables of refinements are in the map when the refinement is evaluated. When it is the case, we call transitions with such refinements *well-defined*.

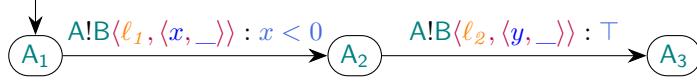
► **Example 34** (Well-defined transition). Considering the RCS in Figure 5. In the RCFSM of A , the (local) state A_2 is only accessible with a transition $A_1 \xrightarrow{A!B<ell_1, <x, __> : x < 0} A_2$. Therefore, any global state $\langle\langle A_2, B_{\{1,2,3\}}, __, M\rangle\rangle$ necessarily contains a preceding transition $A_1 \xrightarrow{A!B<ell_1, <x, __> : x < 0} A_2$. Therefore, x is always in the map M of that state. Therefore, the transition $A_2 \xrightarrow{A!B<ell_2, <y, __> : x < 10} A_3$ is well-defined.

We can now conclude our analysis technique: consider a target transition t with refinement r that is self-independent (it does not modify the variables of its refinement) and well-defined. If all transitions that modify the free variables of r can guarantee (via their refinement) that the modification they do is correct with respect to r , then we can safely remove r .

► **Theorem 35** (Correctness of refinement elision). *Given an RCS R containing an RCFSM $M = \langle Q, C, q_0, \mathbb{A}, \delta \rangle$, and $t = s_i \xrightarrow{p \dagger q(m):r} s'_i \in \delta$, a well-defined self-independent transition. Let $t' = s_i \xrightarrow{p \dagger q(m):T} s'_i$; $\delta' = \delta \setminus \{t\} \cup t'$; $M' = \langle Q, C, q_0, \mathbb{A}, \delta' \rangle$; and R' be R where M is replaced with M' . If, for each transition $t_w = _ \xrightarrow{_! \langle _ \rangle : r_w} _$ in $\bigcup_{x \in \text{fv}(r)} \mathbb{T}_x$, for all map M , $M \models r_w$ entails $M \models r$, then there exists a bisimulation relating the states of R' and R .*

Proof. Proving each direction of the bisimulation is direct (see the proof in [35]). \blacktriangleleft

► **Example 36** (Application of Theorem 35). The following RCFSM, where $x < 10$ is removed, is a valid replacement for $\mathcal{A}(G_s \upharpoonright_A)$ in $\mathcal{S}(G_s)$.



6.2 Application to RMPST Protocols

The above subsection explains how to remove some redundant refinements in RCS. In this subsection, we intend to do the same, focusing on RMPST instead of RCS.

Our goal is the following: we are given an RMPST G , and we would like to remove one of its refinement (which we call the *target* refinement r). For the sake of simplicity, in this section, we assume all labels are uniquely used. For the general case, we can simply uniquely rename redundant labels. Overall, the roadmap for this subsection is to show that given the type G' , which is G where r is replaced by T , G and G' behave similarly, i.e. the RCS they generate are bisimilar. To achieve this, we show that Theorem 35 applies to $\mathcal{S}(G)$ and $\mathcal{S}(G')$. Therefore, the main point is finding conditions on RMPST that ensures hypothesis of Theorem 35 holds; we have to verify the following items:

1. all transitions our refinement depends on should entail the refinement itself;
2. the transition that carries the refinement must be well-defined. Since variables cannot be removed from the map, the first occurrence of the target transition must respect the domain condition. Therefore, for this step, we can ignore recursion.

The main difference with automata is that, in types, we have *communications*, which possibly contains choices with multiple branches; and we our goal is to remove the refinement of one of those branches. Therefore, we first introduce *steps* of a communication, i.e. given a choice, what are the possible choices it can take. We then extend this to types. We show that steps in a type correspond to transitions in the automata of that type.

► **Example 37** (Step). The type $G_y = A \rightarrow B \{l_2(y : \text{int} \models x < 10).\text{end}\}$ has the step $A \rightarrow B(l_2, y) \models x < 10$. Since G_y occurs in one of the branches of G_s (from the introduction of this section), this step *occurs* in G_s .

Given this notion of steps occurring in a type that is analogous to transitions in the RCFSM of that type, we can now focus on the conditions of Theorem 35. Therefore, we have to characterise what corresponds to *well-defined transitions* in a type. Since transitions (in RCS) and steps (in types) are analogous, we introduce *well-defined steps* in a type. We recall that, in a RCS, a transition is well-defined if the free variables of the refinement it carries are always known when the transition is fired. Since variables are never removed from the map, we can focus on the first occurrence of the transition. So far, we do not have a notion of *run* for a type. Therefore, we first define an *happens-before* relation in RMPST, and we use this relation to define *well-defined steps* as steps that contain a refinement which free variable are all exchanged in a communication that *happens-before* the step we consider. With those two definitions, we can finally prove that a well-defined step in a type corresponds to a well-defined transition in the corresponding RCS.

► **Example 38** (Well-defined step in a type). Consider G_s and G_y as in Example 37. The step $A \rightarrow B(l_2, y) \models x < 10$ is well-defined. Indeed, $\text{fv}(x < 10) = \{x\}$, $G_s < G_y$, and G_s contains a branch that sends x and which continuation contains G_y .

We can finally proceed to the overall goal of this section: showing that the type with and without the target refinement behave similarly. Thanks to the above lemmata, we simply have to target a refinement with the appropriate conditions and apply Theorem 35.

► **Theorem 39** (Static elision of redundant refinements in types). *Given two a global types G and $G_s = p \rightarrow q\{\ell_i(x_i : S_i \models r_i).G_i\}_{i \in I} \in G$, such that, for one $t \in I$, $p \rightarrow q(\ell_t, x_t) \models r_t$ is a well-defined step with $x_t \notin \text{fv}(r_t)$. Let $\ell_{t'} = \ell_t$, $x_{t'} = x_t$, $S_{t'} = S_t$, $r'_{t'} = \top$, $G_{t'} = G_t$, $G_{s'} = p \rightarrow q\{\ell_i(x_i : S_i \models r_i).G_i\}_{i \in I \setminus \{t\} \cup \{t'\}}$; and G' be G where G_s is replaced with $G_{s'}$. If, for all steps, $r \rightarrow s(_, x_w) \models r_w$ occurring in G (for each $x \in \text{fv}(r)$), $M \models r_w$ entails $M \models r$ (for all M), there exists a bisimulation between the states of $\mathcal{S}(G)$ and those of $\mathcal{S}(G')$.*

Proof. We prove this by showing that Theorem 35 applies to $\mathcal{S}(G)$ and $\mathcal{S}(G')$. The proof is provided [35]. ◀

► **Example 40** (Application of Theorem 39). Given G_s as in Example 37 and G'_s as follows (notice the second refinement is replaced by \top), G_s and the following G'_s have the same behaviour:

$$G'_s = A \rightarrow B \{\ell_1(x : \text{int} \models x < 0).A \rightarrow B \{\ell_2(y : \text{int} \models \top).\text{end}\}\}$$

7 Implementation

In the previous section, we introduced an instance of our framework: a system that accommodates refinements using a decentralised verification mechanism. In this section, we follow up on this example with an implementation, based on Rumpsteak, of this system.

Rumpsteak [7] is a framework to write Rust programs according to an MPST specification. The framework is divided into two parts:

- (i) a runtime library that provides primitives to write asynchronous programs in Rust; and
- (ii) a tool (`rumpsteak-generate`) to generate skeleton Rust files from specification files (i.e. from global types), in two steps.

Working with Rumpsteak takes two manual steps. The user specifies (step 1) the protocol in a global type (written as Scribble files [39], see Figure 6a). This global type is automatically projected using ν Scr [15] and the projected types are used to generate skeleton Rust files (see Figure 6b). The generated Rust code contains Rust types that encode local types (e.g. the type for A is shown in Line 1 in Figure 6b). The user then manually implements (step 2) the process of each participant, following their type (Line 7), using provided communication primitives (Line 13). Rumpsteak relies on Rust’s typechecker to ensure the consistency of the implementation. For the sake of clarification where needed, we call *Vanilla Rumpsteak* the framework without refinements (i.e. as presented in [7]), and *Refined Rumpsteak* the framework modified to accommodate refinements.

In this section, we explain the main differences between Vanilla and Refined Rumpsteak: we introduce refinements in the types used in the runtime library, we modify the program generation step accordingly, and we introduce tools that ensure the localisation conditions are met (Definition 29 in Section 5). The overall workflow is presented in Figure 7. We conclude this section by measuring the overhead induced by the refinement w.r.t. Vanilla Rumpsteak and the time needed for asserting the localisation conditions.

```

1 (*# RefinementTypes #*)
2
3 global protocol PlusMinus
4 (role A, role B, role C)
5 {
6   Secret(n: int) from A to B;
7   rec Loop {
8     Guess(x: int) from C to B;
9     choice at B {
10       More(x: int {x < n}) from B to C;
11       continue Loop;
12     } or {
13       Less(x: int {x > n}) from B to C;
14       continue Loop;
15     } or {
16       Correct(x: int {x = n}) from B to C;
17     }}}

```

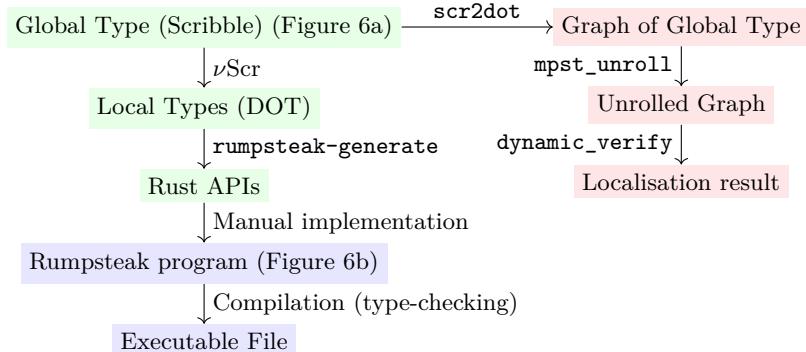
(a) ν Scr description of the guessing game protocol.

```

1 type PlusMinusA =
2 Send<B, 'n',
3 Secret,
4 Tautology::<Name, Value, Secret>,
5 Constant<Name, Value>, End>;
6 // ...
7 async fn a(role: &mut A)
8 -> Result<(), Box<dyn Error>> {
9   try_session(role,
10   HashMap::new(),
11   |s: PlusMinusA<'_, _>| async {
12     let s =
13     s.send(Secret(10)).await?;
14     return Ok((((), s))
15   })
16   .await
17 }

```

(b) Rust type and implementation of participant A of the guessing game protocol. The handwritten code (Line 7 to Line 17) is the same than with Vanilla Rumpsteak.

Figure 6 Implementation of the guessing game using Rumpsteak.**Figure 7** Workflow of Rumpsteak. Green nodes represent steps that already existed in Vanilla Rumpsteak and that have been adapted to accommodate for refinements, red nodes represent new steps, and blue nodes represent unmodified steps. The three new steps (`scr2dot`, `mpst_unroll`, and `dynamic_verify`) verify the conditions mentioned in Definition 29.

7.1 Refinement Implementation

Modifications to the Rumpsteak Library. In order to accommodate for refinements, we have to introduce new elements in to the Rumpsteak's encoding of local types. Consider the local type of participant A introduced in Example 21 $B \oplus \{secret(n : int \models \top).end\}$: Rumpsteak now has to take into account the name of the variable sent (`n`), and the refinement attached to the transition (`T`). Consider the type declaration in Line 1 to Line 5, Figure 6b. Compared to Vanilla Rumpsteak, we introduce '`n`', a const generic⁵, that carries the name of the variable sent (Line 4). Regarding the refinement, we introduce `Tautology::<Name, Value, Secret>`,

⁵ <https://github.com/rust-lang/rfcs/blob/master/text/2000-const-generics.md>

which represent the refinement T . The generic parameters are used to specify the type of variable names (`chars` in our case) and values (`i32`) as well as the label of the message (`Secret`). We modified `vScr` and `rumpsteak-generate` to generate skeleton files (the content of the file up to Line 5). Rumpsteak provides a set of available refinements, and additional ones can be written ad-hoc (for specific needs). To add an ad-hoc refinement, the user simply implements the trait `Predicate` (which extends `Default`), which requires a method `check` that asserts whether the predicate holds. For instance, the `check` function of `Tautology` simply returns `true`.

Verification of the Conditions for Decentralised Refinement Assertion. As we explained in Section 5, to make sure that refinements can be verified in a decentralised way, we require to check that variables needed for the refinements are located correctly (Definition 29). To perform this verification, we implemented new tools for the Rumpsteak framework (in red in Figure 7).

Our tools:

- (i) convert the global type into a graph (`scr2dot`);
- (ii) unroll the loops once to precisely capture variables initialisations (`unroll_mpst`); and
- (iii) localise variables on the unrolled graph (`dynamic_verify`).

The core part of this verification, `dynamic_verify`, finds variables locations with simple inference rules written in Datalog. We use the `crepe` library [40] which provides a Datalog DSL for Rust. We provide more details on the algorithm in [35].

Limitations. The current implementation makes extensive use of the Rust feature *const generics*⁹ which introduces a limited form of dependent types in Rust. It allows to use constant values in types. As of today, only some *basic types* can be used as const generics, in particular `chars` and the various integer types. We use such const generics to encode informations about the variables into the types: for instance, the predicate `x < 5` would have the type `LTnConst<L, 'x', 5>`, where the '`x`' and the 5 are const generics.

For readability, we choose to set variables to `chars`, meaning that in the current implementation, we can only accommodate a limited number of distinct variables. Should more be needed, one could easily modify our implementation to replace them with `u64`, which allows 2^{64} variables names. Similarly, we only consider `i32` as message payloads. Should different types of messages be needed, they could be encoded in an `enum`.

Finally, the static elision optimisation (Section 6) is not implemented.

7.2 Runtime and Localisation Benchmarks

We evaluate how Rumpsteak with refinements performs with respect to Rumpsteak without refinements. First, we measure the runtime of our analysis tool which verifies the two conditions in Definition 29 (`scr2dot`, `unroll_mpst` and `dynamic_verify`). Although not a runtime cost, and while we expect this analysis to be possibly expensive, we would like to ensure that it is still practical for test cases from the literature. Secondly, we evaluate the runtime overhead of adding refinements with respect to Rumpsteak without refinements.

Setup and Benchmark Programs. We evaluate the performance of Rumpsteak with refinement with benchmarks. Most of them are taken from the literature (Table 1). This set of program contains various micro-benchmarks with a variety of combination of properties (whether the protocol is binary or multiparty, contains recursivity or choice). Notice that protocols that contain recursivity with no choice (e.g. *simple auth* are infinite). Therefore,

Table 1 The set of micro benchmarks together with their characteristics. “MP” denotes a multiparty protocol, “Rec” the presence of recursion, and “Choice” the presence of choice.

Name	MP	Rec	Choice
① simple adder [21]	no	no	no
② travel_agency [23]	no	no	yes
③ ping pong [42]	no	yes	no
④ simple auth.	no	yes	yes
⑤ ring max	yes	no	no
⑥ three_buyers [19]	yes	no	yes
⑦ plus or minus	yes	yes	yes

such protocols are only measured in the variable localisation paragraph. Also, where it applies, protocols were modified in order to add relevant refinements; such modifications are listed below. By default, we add **Tautology** predicates (Section 7.1). The tests were performed on a machine running Ubuntu 22.04.1 LTS x86_64 (kernel 5.15.0-60) with an Intel i7-6700 processor (4 cores, 8 threads running at 4GHz maximum) and 16GB of memory⁶. We compare Rumpsteak with refinement vs. Vanilla Rumpsteak. For a comparison between Vanilla Rumpsteak and other libraries, see [7, Figure 6].

Added Refinements & Protocol Modifications. Some benchmarks from the literature were adapted in order to accommodate refinements. In addition, we introduce three benchmarks. Those benchmarks are close to examples from the literature, adapted to better highlight refinements.

simple adder: This example is adjusted from the *Adder* ([21]) protocol, but we remove the choice of operation in order to increase the benchmark diversity;

ping pong: In [42], some of the loops were statically unrolled, and the protocol contained a choice to exit. Ours is equivalent to an infinite *PingPong*₁ in [42].

simple authentication: This example is a binary example of an authentication protocol (e.g. OAuth [31]). The added refinements enforce that access is granted if and only if the given password is correct.

ring max: A multiparty protocol where participants receive a value from their predecessor (except for the initial participant), and forward an other value to their successor (the final participant forwards it to the initial one). Refinements ensure that the value forwarded is greater than or equal to the value received.

plus or minus: An implementation of our running example.

Static Analysis of Variable Locations. Table 2 shows the decentralised verification time cost for each refined global label. As shown in Figure 7, this static analysis is performed with three tools. The results shown account for the whole pipeline, and were measured over 50 samples, with 10 warmup runs (excluded from the measurements). Overall, the runtime for variable localisation is stable (around 5.6ms). We suspect that, for graphs with a low number of states, the runtime is dominated by the accesses to the file.

⁶ The micro-benchmarks are not memory intensive. The memory size is not a limiting factor. However, the benchmarks seem to be dominated by the startup time, which includes memory access time.

Table 2 Benchmark of the localisation analysis (Red branch in Figure 7). $|S|$ denotes the number of states of the graph of the protocol; $|U|$ denotes the number of states after unrolling the recursion loops once; and $|V|$ denotes the number of variables in the protocol. $|S|$, $|U|$ and $|V|$ are computed manually to give an insight on how protocols compare. et is the execution time, measured by the benchmark (in ms).

	$ S $	$ U $	$ V $	$et (\mu \pm \sigma)$
①	4	4	3	5.5 ± 0.2
②	7	7	6	5.5 ± 0.2
③	2	4	1	5.5 ± 0.2
④	6	11	3	5.6 ± 0.2
⑤	8	8	7	5.7 ± 0.2
⑥	10	10	7	5.6 ± 0.2
⑦	4	19	2	5.6 ± 0.2

Table 3 Evaluation of the runtime overhead due to the addition of refinements in Rumpsteak. p is the MWU p -value, m is the baseline median runtime and m_r is the median runtime with refinements when applicable ($p < 0.05$). All times are in ms.

	p	m	m_r
①	0.00	0.7	0.8
②	0.11	0.8	N/A
④	0.29	0.8	N/A
⑤	0.17	0.8	N/A
⑥	0.68	0.7	N/A
⑦	0.04	0.7	0.8

Runtime Overhead of Refinement Feature. Our second set of benchmarks aims to measure the overhead of runtime refinement verification with respect to the original Rumpsteak framework. We are expecting Rumpsteak with refinements to be slower than the original Rumpsteak, due to the additional cost of evaluating refinements. This benchmark has two objectives: first, to find out whether there is an actual, statistically significant, overhead; and second, if so, estimate this overhead. To measure this overhead, we only consider the protocols that terminate from the benchmark set.

To fulfil the first objective, we use a Mann-Whitney U test (MWU). We used MWU as it is a non-parametric test, and our runtime distributions do not follow a normal distribution, which prevents us to do simpler analysis. As MWU is sensitive to the number of samples, we run each benchmark 30 times, on both the original Rumpsteak and Rumpsteak with refinements. We perform the MWU test on the collected 30 samples, preceded by 10 iterations to warm the system up. Our hypothesis for the MWU test are the following:

$$\begin{aligned} H_0: & \text{The distributions of runtimes with and without refinements are identical.} \\ H_1: & \text{The distributions of runtimes with and without refinements are distincts.} \end{aligned}$$

The p -values obtained from the MWU test are reported in the first column of Table 3. We also report the baseline (Rumpsteak without refinements) median run time (over the 30 runs) in the second column of the table. Most often, the overhead is not significant ($p \geq 0.05$) and H_0 can not be rejected. When the overhead is statistically significant, we also report the median runtime (over the 30 runs) of Rumpsteak with refinements in the third column. With our set of microbenchmarks, in most cases we cannot distinguish Rumpsteak with refinement from Rumpsteak without refinements. We suspect Rumpsteak runtime is dominated by communications and context switching. However, as our refinements can be arbitrarily complex, specific instances could show real slowdown due to refinement evaluation.

8 Related Work and Conclusion

Design-by-Contract for (Multiparty) Session Types. In binary session types, [37] introduces contracts for binary sessions, and provides an analysis tool which verifies whether a given program complies with its associated contract. The verification is done with symbolic execution. Compared to this paper, we address multiparty sessions. Besides, our framework is more generic (specific instances could be based on symbolic execution, but we can also accommodate other verification methods). Bocchi et al. [4] present a variant of MPST that allows predicates on exchanges, that must hold for a typed process to take transitions. The main difference with our work is that their approach focuses on *correctness by construction*, i.e. they accept only correct protocols, while we can accept protocols that fail, and we simply prevent them to generate incorrect traces. More precisely, the authors statically ensure that there is a satisfiable path, which prevents some valid runs to be accepted. For instance, consider the following type:

$$\text{A} \rightarrow \text{B} \{ \ell_1(x : \text{int} \models x < 10). \text{B} \rightarrow \text{A} \{ \ell_2(y : \text{int} \models x > y \wedge y > 6). \text{end} \} \}$$

This type would be rejected in [4] since if **A** sends $x = 5$ (which is allowed by $x < 10$), then there is no y that satisfies $5 > y \wedge y > 6$. By rejecting this, they also reject all possibly valid runs (e.g. if **A** sends $x = 9$ and **B** replies with $y = 7$). A follow-up on this work is [3] which introduces local states, i.e. the authors allow participants to have local variables, which can be updated during process execution. The session types reflect those elements and contain predicates on exchanged variables and local variables.

With respect to these two papers, our criteria for the validity of refinements (expressed as a property of the generated trace) is decoupled from the semantics of the model. This approach allows us to be more flexible than enforcing statically the refinements, and to lower the cost of adopting refinements, in particular to retrofit refinements into existing systems. For instance, using our framework, one can simply use the centralised semantics at first, which is very expressive, without having to prove the correctness of the implementation. In a second step, users can then develop different verification or analysis techniques which can be plugged-in transparently. For instance, switching from Vanilla Rumpsteak to Refined Rumpsteak does not involve changes in the implementation, as the modifications do not happen in the programming interface. Also, compared to these papers, our framework is not bound to MPST only, and provide an actual implementation of our framework.

Design-by-Contract in Choreography Automata. Choreography Automata (CA) are graphs that represent the global behaviour of a concurrent system. The behaviour of individual participants is obtained by *projecting* well-formed CA, i.e. erasing all actions that do not concern a given participant. The result is a FSM which, after determinising and minimising, is used as a CFSM. The projection of all participants leads to a CS. Notice that CA accept some protocols that would be rejected by MPST, and vice-versa.

Gheri et al. [16] study the verification of CA with assertions. Their work and ours are distinct with respect to the following aspects:

- (i) the communication semantics;
- (ii) the choices;
- (iii) the logic for predicates; and
- (iv) the implementation presented in [16] is limited to CA without assertions (i.e., the design-by-contract approach was not implemented and left as their future work).

Regarding Item i, Gheri et al. [16] defines choreography automata with *synchronous* communication semantics, while the one we developed in this work is asynchronous. Gheri et al. [16, Section 7] discusses asynchronous semantics but it remains future works.

Regarding Item ii, we are constrained by the syntax of RMPST, in which choices can only happen between two selected participants, while choreography automata accept protocols with choices where a (single) participant **A** sends to multiple receivers (**B** and **C**) [16, Definition 4.15]. Explicit connections [22] is an extension of MPST that accommodates with choices with multiple receivers.

Regarding Item iii, we kept our refinement logic abstract, while it is fixed in choreography automata, with a form of first order logic. Besides, predicates are handled differently in both frameworks as well: Gheri et al. [16] require choreography automata to be *history-sensitive* [4], a definition which serves a similar purpose to our definition of *variable localisation* (Section 5 and [35]), which constrains our decentralised semantics. Our centralised semantics (Definition 10) is not constrained by variable localisation. For instance, the RMPST $A \rightarrow B \{ \ell_1(x : \text{int} \models T). C \rightarrow D \{ \ell_2(y : \text{int} \models x = y). \text{end} \} \}$ produces valid traces with our centralised semantics, while the corresponding choreography automata would be rejected.

Besides, our work introduces a general framework that can accommodate refined CA in addition to RMPST. We show [35] a possible way to do so.

Implementations of Refinements in MPST. Neykova et al. [29] develop an F# library for static verification of MPST with refinements. They present a compiler plugin which uses an SMT solver (Z3) to statically verify some refinements. They use a notion of similar to our variable localisation criterion (which they call *variable knowledge*), and a variant of CFSM with refinements that is similar to ours. In their work, refinements that are statically asserted by the SMT solver are pruned in the CFSM, while the rest of refinements are kept in the CFSM and are dynamically checked. Similarly, [41, 42] develop a framework for multiparty session types with refinements in F*. They delegate the management of refinements to F* type system (which internally uses an SMT solver). They define refinements on global types, which are then projected onto local types. They show that a global type and its projection are trace equivalent. Those two works focus on the *implementation* of MPST with refinements. [29] does not focus on the theory of refinements and the theory developed in [42] is tightly coupled to F*. For instance, they do not present a *correctness* criterion such as *valid refined traces* we present. Contrary to both works, our correctness criteria (based on valid refined traces) is *decoupled from* (i.e. independent of) any target type theory, programming language or model of computation: we only require an LTS labelled with actions. Besides, the logic used for refinements is also a parameter of our framework, and users could use alternatives, leading to a greater expressivity of our framework.

The main syntactical difference between our RMPST and those developed in [42] is that we attach refinements to the messages of the protocol, while [42] attach refinements to the payload value. This is due to a different approach: correctness in [42] is related to payload types being inhabited while our criteria of correctness (developed in Definition 7) relies on actions being allowed. In binary linear logic-based session types, [9] study the metatheory of binary session types with arithmetic refinements. In particular, they focus on the type equality, showing that added refinements make the type equality undecidable (they provide a sound but incomplete algorithm for type equality). [10] also implement a library for session types with refinements, although it only accounts for arithmetic refinements.

Other Related Works. There are various papers on the dynamic verification of MPST. For instance [2] present a framework that allows for both static and dynamic verification of MPST. This paper introduces a theory for (dynamically) monitoring assertions on messages (i.e. the equivalent of our refinements). Furthermore, the authors introduce theoretical tools (bisimulations) to relate monitored processes with correct unmonitored processes. This paper, however, suffers a few limitations. First, it focuses on *monitorable* types (which intuitively correspond to types satisfying our conditions for decentralised verification Definition 29). Second, it focuses on dynamic verification of assertions. The paper is compatible with statically verified processes (which allows turning off the dynamic monitoring), but it does not present techniques for static verification in itself.

On the other hand, our paper takes a different approach, by decoupling the correctness criterion from the verification technique. This allows us to have a more general framework (our framework accept types that are not localisable/monitorable, although not all semantics can accommodate those), as well as to develop static verification techniques.

In Rust, the `refinement` crate [11] provides refinement data types. Their approach of refinements is similar to ours, with a `Predicate` trait that provides a method to perform the predicate verification (at runtime). Refinement data types have also been implemented in multiple languages (e.g. F^{*}, Haskell [36], etc.). On the practical side, we can note the similarities between typestates and session types [20]. [14] implements typestates in Rust with a DSL to verify protocol conformance. While Rumpsteak does not use their library, it internally uses similar constructs.

Regarding implementations of session types in Rust, there are several frameworks beside Rumpsteak. [25] first integrate binary session types in Rust, but their implementation suffers a few drawbacks (see [26, Section 3] for a detailed explanation). Sesh [26] and Ferrite [6] are two Rust libraries for *binary* session types, and they implement synchronous and asynchronous ones, respectively. MultiCrusty [27] implements synchronous MPST on top of Sesh, with a mesh of binary sessions. Compared to MultiCrusty, Rumpsteak implements directly MPST instead of wrapping them into binary sessions, and focuses on asynchronous MPST. None of the aforementioned tools develops refinements. It would be an interesting future work to apply our criteria to extend their tools with refinements.

Finally, we note the proximity between (MP)ST with refinements and dependent (MP)ST. For instance, [33] introduce a session type calculus with label-dependency (their approach does not explicitly account for payload value refinement). Other approaches exist, for instance, an intuitionistic linear logic-based type theory for building value-dependent session types [34], and separation logic-based work for reasoning about session types [17].

Future Work While, in our work, we consider MPST with payloads (some variants only consider messages with labels), we restrict our MPST with a single payload (i.e. *monadic* MPST, where each message carries a single value). The extension to polyadic MPST, where a message can carry multiple values, is straightforward, by adapting the RCS rules (GRSND and GRREC, Definition 14).

We presented two optimisations, in order to illustrate the flexibility of our theoretical framework. Regarding the decentralised verification (Section 5), there is room for an extension, e.g. with specific domains (i.e. some class of protocols with specific refinements). Regarding the static elision of redundant refinements, we envision improving the technique with use of SMT solvers could be promising. The main difficulty lies in asynchronous communications: one would need to consider all possible message orderings before solving constraints.

References

- 1 Franco Barbanera, Ivan Lanese, and Emilio Tuosto. Choreography automata. In Simon Blidze and Laura Bocchi, editors, *Coordination Models and Languages - 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15–19, 2020, Proceedings*, volume 12134 of *Lecture Notes in Computer Science*, pages 86–106. Springer, 2020. doi:[10.1007/978-3-030-50029-0_6](https://doi.org/10.1007/978-3-030-50029-0_6).
- 2 Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. Monitoring networks through multiparty session types. *Theoretical Computer Science*, 669:33–58, 2017. doi:[10.1016/j.tcs.2017.02.009](https://doi.org/10.1016/j.tcs.2017.02.009).
- 3 Laura Bocchi, Romain Demangeon, and Nobuko Yoshida. A Multiparty Multi-Session Logic. In *7th International Symposium on Trustworthy Global Computing*, volume 8191 of *LNCS*, pages 111–97. Springer, 2012.
- 4 Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. A Theory of Design-by-Contract for Distributed Multiparty Interactions. In Paul Gastin and François Laroussinie, editors, *CONCUR 2010 - Concurrency Theory*, Lecture Notes in Computer Science, pages 162–176, Berlin, Heidelberg, 2010. Springer. doi:[10.1007/978-3-642-15375-4_12](https://doi.org/10.1007/978-3-642-15375-4_12).
- 5 Daniel Brand and Pitro Zafiropulo. On Communicating Finite-State Machines. *Journal of the ACM*, 30(2):323–342, April 1983. doi:[10.1145/322374.322380](https://doi.org/10.1145/322374.322380).
- 6 Ruofei Chen and Stephanie Balzer. Ferrite: A Judgmental Embedding of Session Types in Rust, 2021. (repository is found at <https://github.com/ferrite-rs/ferrite>). arXiv:2009.13619.
- 7 Zak Cutner, Nobuko Yoshida, and Martin Vassor. Deadlock-free asynchronous message reordering in rust with multiparty session types. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP ’22*, pages 246–261, New York, NY, USA, April 2022. Association for Computing Machinery. doi:[10.1145/3503221.3508404](https://doi.org/10.1145/3503221.3508404).
- 8 Gérard Cécé and Alain Finkel. Verification of programs with half-duplex communication. *Information and Computation*, 202(2):166–190, November 2005. doi:[10.1016/j.ic.2005.05.006](https://doi.org/10.1016/j.ic.2005.05.006).
- 9 Ankush Das and Frank Pfenning. Session Types with Arithmetic Refinements. In Igor Konnov and Laura Kovács, editors, *31st International Conference on Concurrency Theory (CONCUR 2020)*, volume 171 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:18, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:[10.4230/LIPIcs.CONCUR.2020.13](https://doi.org/10.4230/LIPIcs.CONCUR.2020.13).
- 10 Ankush Das and Frank Pfenning. Rast: A Language for Resource-Aware Session Types. *Logical Methods in Computer Science*, Volume 18, Issue 1, January 2022. doi:[10.46298/lmcs-18\(1:9\)2022](https://doi.org/10.46298/lmcs-18(1:9)2022).
- 11 Brady Dean and Joey Ezechiëls. refinement crate, 2021. (repository is found at <https://github.com/2bdkid/refinement>).
- 12 Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty Session Types Meet Communicating Automata. In *21st European Symposium on Programming*, volume 7211 of *LNCS*, pages 194–213. Springer, 2012.
- 13 Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types. In *40th International Colloquium on Automata, Languages and Programming*, volume 7966 of *LNCS*, pages 174–186, Berlin, Heidelberg, 2013. Springer. doi:[10.1007/978-3-642-39212-2_18](https://doi.org/10.1007/978-3-642-39212-2_18).
- 14 José Duarte and António Ravara. Retrofitting Typestates into Rust. In *25th Brazilian Symposium on Programming Languages*, pages 83–91, Joinville Brazil, September 2021. ACM. doi:[10.1145/3475061.3475082](https://doi.org/10.1145/3475061.3475082).
- 15 Francisco Ferreira, Fangyi Zhou, Simon Castellan, and Benito Echarren. NuScr, 2019. URL: <https://github.com/nuscr/nuscr>.

- 16 Lorenzo Gheri, Ivan Lanese, Neil Sayers, Emilio Tuosto, and Nobuko Yoshida. Design-By-Contract for Flexible Multiparty Session Protocols. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:28, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:[10.4230/LIPIcs.ECOOP.2022.8](https://doi.org/10.4230/LIPIcs.ECOOP.2022.8).
- 17 Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. Actris: Session-type based reasoning in separation logic. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–30, January 2020. doi:[10.1145/3371074](https://doi.org/10.1145/3371074).
- 18 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *ACM SIGPLAN Notices*, 43(1):273–284, January 2008. doi:[10.1145/1328897.1328472](https://doi.org/10.1145/1328897.1328472).
- 19 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty Asynchronous Session Types. *Journal of the ACM*, 63(1):9:1–9:67, March 2016. doi:[10.1145/2827695](https://doi.org/10.1145/2827695).
- 20 Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. Type-safe eventful sessions in java. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP’10, pages 329–353, Berlin, Heidelberg, June 2010. Springer-Verlag.
- 21 Raymond Hu and Nobuko Yoshida. Hybrid Session Verification Through Endpoint API Generation. In Perdita Stevens and Andrzej Wąsowski, editors, *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, pages 401–418, Berlin, Heidelberg, 2016. Springer. doi:[10.1007/978-3-662-49665-7_24](https://doi.org/10.1007/978-3-662-49665-7_24).
- 22 Raymond Hu and Nobuko Yoshida. Explicit Connection Actions in Multiparty Session Types. In Marieke Huisman and Julia Rubin, editors, *Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science, pages 116–133, Berlin, Heidelberg, 2017. Springer. doi:[10.1007/978-3-662-54494-5_7](https://doi.org/10.1007/978-3-662-54494-5_7).
- 23 Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-Based Distributed Programming in Java. In Jan Vitek, editor, *ECOOP 2008 – Object-Oriented Programming*, Lecture Notes in Computer Science, pages 516–541, Berlin, Heidelberg, 2008. Springer. doi:[10.1007/978-3-540-70592-5_22](https://doi.org/10.1007/978-3-540-70592-5_22).
- 24 International Telecommunication Union. Z.120 : Message Sequence Chart (MSC), February 2011.
- 25 Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. Session types for Rust. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, pages 13–22, Vancouver BC Canada, August 2015. ACM. doi:[10.1145/2808098.2808100](https://doi.org/10.1145/2808098.2808100).
- 26 Wen Kokke. Rusty Variation: Deadlock-free Sessions with Failure in Rust. *Electronic Proceedings in Theoretical Computer Science*, 304:48–60, 2019. (repository is found at <https://github.com/wenkokke/sesh>). doi:[10.4204/eptcs.304.4](https://doi.org/10.4204/eptcs.304.4).
- 27 Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. Stay Safe Under Panic: Affine Rust Programming with Multiparty Session Types. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:29, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISSN: 1868-8969. doi:[10.4230/LIPIcs.ECOOP.2022.4](https://doi.org/10.4230/LIPIcs.ECOOP.2022.4).
- 28 Bertrand Meyer. Design by Contract. *Advances in Object-Oriented Software Engineering*, pages 1–35, 1991.
- 29 Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. A session type provider: compile-time API generation of distributed protocols with refinements in F#. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pages 128–138, New York, NY, USA, February 2018. Association for Computing Machinery. doi:[10.1145/3178372.3179495](https://doi.org/10.1145/3178372.3179495).
- 30 Davide Sangiorgi. *An Introduction to Bisimulation and Coinduction*. Cambridge University Press, Cambridge ; New York, 2012.

- 31 Alceste Scalas and Nobuko Yoshida. Less is more: multiparty session types revisited. *Proceedings of the ACM on Programming Languages*, 3(POPL):30:1–30:29, January 2019. doi:[10.1145/3290343](https://doi.org/10.1145/3290343).
- 32 Felix Stutz. Asynchronous Multiparty Session Type Implementability is Decidable - Lessons Learned from Message Sequence Charts. In *DROPS-IDN/v2/Document/10.4230/LIPIcs.ECOOP.2023.32*. Schloss-Dagstuhl - Leibniz Zentrum für Informatik, 2023. doi:[10.4230/LIPIcs.ECOOP.2023.32](https://doi.org/10.4230/LIPIcs.ECOOP.2023.32).
- 33 Peter Thiemann and Vasco T. Vasconcelos. Label-dependent session types. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–29, January 2020. doi:[10.1145/3371135](https://doi.org/10.1145/3371135).
- 34 Bernardo Toninho, Luís Caires, and Frank Pfenning. Dependent session types via intuitionistic linear type theory. In *Proceedings of the 13th International ACM SIGPLAN Symposium on Principles and Practices of Declarative Programming*, pages 161–172, Odense Denmark, July 2011. ACM. doi:[10.1145/2003476.2003499](https://doi.org/10.1145/2003476.2003499).
- 35 Martin Vassor and Nobuko Yoshida. Refinements for multiparty message-passing protocols: Specification-agnostic theory and implementation, 2024. Full version on Arxiv.
- 36 Niki Vazou. *Liquid Haskell: Haskell as a Theorem Prover*. PhD thesis, University of California, San Diego, USA, 2016. URL: <http://www.escholarship.org/uc/item/8dm057ws>.
- 37 Jules Villard. *Heaps and Hops*. PhD thesis, Laboratoire Spécification et Vérification, École Normale Supérieure de Cachan, France, February 2011.
- 38 Nobuko Yoshida and Lorenzo Gheri. A Very Gentle Introduction to Multiparty Session Types. In Dang Van Hung and Meenakshi D’Souza, editors, *Distributed Computing and Internet Technology*, Lecture Notes in Computer Science, pages 73–93, Cham, 2020. Springer International Publishing. doi:[10.1007/978-3-030-36987-3_5](https://doi.org/10.1007/978-3-030-36987-3_5).
- 39 Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The scribble protocol language. In Martín Abadi and Alberto Lluch Lafuente, editors, *Trustworthy Global Computing*, pages 22–41, Cham, 2014. Springer International Publishing.
- 40 Erik Zhang. Crepe, 2022. URL: <https://crates.io/crates/crepe>.
- 41 Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Statistically Verified Refinements for Multiparty Protocols. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. doi:[10.1145/3428216](https://doi.org/10.1145/3428216).
- 42 Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Statistically Verified Refinements for Multiparty Protocols. *arXiv:2009.06541 [cs]*, September 2020. arXiv: 2009.06541. arXiv:2009.06541.

Failure Transparency in Stateful Dataflow Systems

Aleksey Veresov¹ 

EECS and Digital Futures, KTH Royal Institute of Technology, Stockholm, Sweden

Jonas Spenger¹ 

EECS and Digital Futures, KTH Royal Institute of Technology, Stockholm, Sweden

Paris Carbone 

EECS and Digital Futures, KTH Royal Institute of Technology, Stockholm, Sweden
Digital Systems, RISE Research Institutes of Sweden, Stockholm, Sweden

Philipp Haller 

EECS and Digital Futures, KTH Royal Institute of Technology, Stockholm, Sweden

Abstract

Failure transparency enables users to reason about distributed systems at a higher level of abstraction, where complex failure-handling logic is hidden. This is especially true for stateful dataflow systems, which are the backbone of many cloud applications. In particular, this paper focuses on proving failure transparency in Apache Flink, a popular stateful dataflow system. Even though failure transparency is a critical aspect of Apache Flink, to date it has not been formally proven. Showing that the failure transparency mechanism is correct, however, is challenging due to the complexity of the mechanism itself. Nevertheless, this complexity can be effectively hidden behind a failure transparent programming interface. To show that Apache Flink is failure transparent, we model it in small-step operational semantics. Next, we provide a novel definition of failure transparency based on observational explainability, a concept which relates executions according to their observations. Finally, we provide a formal proof of failure transparency for the implementation model; i.e., we prove that the failure-free model correctly abstracts from the failure-related details of the implementation model. We also show liveness of the implementation model under a fair execution assumption. These results are a first step towards a verified stack for stateful dataflow systems.

2012 ACM Subject Classification Theory of computation → Operational semantics; Software and its engineering → Checkpoint / restart

Keywords and phrases Failure transparency, stateful dataflow, operational semantics, checkpoint recovery

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.42

Related Version *Extended Version:* <https://arxiv.org/abs/2407.06738> [62]

Funding This work was partially funded by Digital Futures under a Research Pairs Consolidator grant (PORTALS).

1 Introduction

Stateful dataflow systems have seen wide adoption in the modern cloud infrastructure due to their ability to process large amounts of event-based data at ingestion time [24]. Apache Flink [13], for example, is used to power tens-of-thousands of streaming jobs with up to nine billion events per second at ByteDance [48], and several thousand streaming jobs at Uber [25]. An essential aspect of stateful dataflow systems is the recovery from failures, as failures are to be expected in any long-running streaming job [19]. However, failure recovery is non-trivial. For example, simply recovering from a failure by restarting a job from the

¹ Both authors contributed equally to this research.

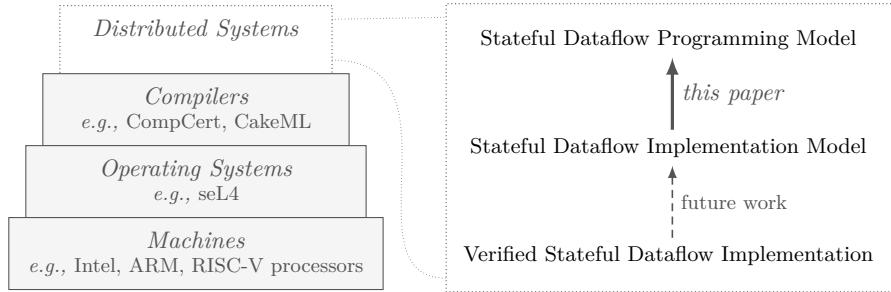


Figure 1 This work in the context of a fully verified stack for distributed programming.

very beginning would discard all progress up to that point, making the recovery prohibitively expensive. To balance the need for efficiency and reliability, stateful dataflow systems have to embrace complex failure recovery protocols. Because of their complexity, the correctness of these recovery protocols is a crucial problem for the reliability of stateful dataflow systems.

In previous work [21, 44, 26], a failure-masking recovery protocol is considered to be correct, if failures can be masked such that the user cannot observe the failures. This property is also known as *failure transparency*, *i.e.*, a user should be able to ignore failures as if they do not occur. Failure transparency has been shown for some distributed systems, such as Durable Functions [8], Reliable State Machines [51], reliable actors (KAR) [61], and serverless microservices (μ 2sls) [30]. As for stateful dataflow, the core mechanism used in Apache Flink’s [13] recovery protocol, namely Asynchronous Barrier Snapshotting (ABS) [11, 12], has been shown to be a correct snapshotting protocol [10]. However, the proof does not reason about failure transparency and its related aspects, such as modelling failures and the recovery from failures, as well as about the equivalence of observed executions. That is, there has been no formal proof that Apache Flink’s entire failure recovery protocol provides failure transparency. Furthermore, the literature lacks a formal definition of failure transparency for systems described with distinct failure-related rules using small-step operational semantics, a widely-used method for defining program execution in programming languages theory.

An important approach for ensuring reliability and correctness is machine-checked formal verification, *i.e.*, proving that a system implements its specification. There is well-known prior work on verified compilers [41, 33, 53], operating systems [31], as well as processors [16, 29, 55] (Figure 1, left). However, there is an apparent lack of verified distributed systems, particularly, there is no verified stateful dataflow system. We believe that it is essential to address this gap in order to prevent disastrous outages of distributed infrastructure as known today.

This work is a first step towards the grand goal of providing a fully verified reliable stack for distributed programming, as shown in Figure 1. It addresses the highlighted gap by: (1) providing a definition of failure transparency, (2) formalizing a stateful dataflow system as a model in small-step operational semantics, under the assumptions of crash-recovery failures and FIFO-ordered channels, and (3) formally proving that the model permits abstracting from failures, *i.e.*, that it is failure transparent. Our definition of failure transparency is based on *observational explainability*, a property which, informally, says that the *explainable* implementation model generates the same observable output as is possible in the *explaining* abstract model. Using this property, a system is defined as failure transparent if the whole system is observationally explainable by its explicitly separated failure-free part. Finally, we prove that our formal model of a stateful dataflow system based on Asynchronous Barrier Snapshotting [13, 10] is failure transparent. This abstraction from failures is designed to serve the end users of the modelled system with less interest in its implementation details.

Contributions. In summary, this paper makes the following contributions.

- We provide the first small-step operational semantics of the *Asynchronous Barrier Snapshotting* protocol within a stateful dataflow system, as used in Apache Flink (Section 4).
- We provide a novel definition of *failure transparency* for programming models expressed in small-step operational semantics with explicit failure rules and the intuitions behind it (Section 5). It is the first attempt to define failure transparency in the context of stateful dataflow systems.
- We prove that the provided implementation model is failure transparent and guarantees liveness (Section 6).
- We provide a mechanization of the definitions, theorems, and models in Coq.²

Outline. Section 2 introduces background on failures, distributed systems, stateful dataflow, as well as some basic notation used throughout this paper. Section 3 informally introduces the stateful dataflow programming model and failure recovery via the Asynchronous Barrier Snapshotting (ABS) protocol. Section 4 provides a small-step operational semantics of a stateful dataflow system based on ABS. Section 5 defines failure transparency and observational explainability for programming models expressed in small-step operational semantics. Section 6 proves that the implementation model is failure transparent. Section 7 discusses related work, and Section 8 concludes this paper.

2 Background

2.1 Failures in Distributed Systems

A distributed system is a system of many processes communicating over a network [9]. The kind of distributed systems which are related to this work are event-based processing systems such as stateful dataflow systems [19, 67, 13, 52, 69, 60]. Failures within such systems are expected to happen, due to their typical large scale and longevity [19]. However, failures are notoriously hard to deal with within distributed systems. For this reason, *failure transparency* is a necessary abstraction, as it enables the user to abstract from failures. Failure transparency as a general concept, and failure recovery protocols are both well-studied topics in distributed systems [63, 65, 40, 44, 43, 26, 21]. Moreover, failure transparency has seen an increase in interest within the programming languages community in recent years [8, 30, 51, 61]. The goal of failure recovery is to provide automatic system means to recover from system failures, in ways which the system user may or may not notice. In contrast, the goal of failure transparency is to provide an abstraction of the system, such that the abstraction hides the internals of failures and failure recovery, masking the failures from the user [26]. For this reason, failure transparency greatly simplifies the programming model to the benefit of the end user.

2.2 Stateful Dataflow and Apache Flink

Stateful dataflow systems, sometimes also called stream processing or dataflow streaming systems, such as Apache Flink [13], have become ubiquitous for real-time processing of large amounts of data [48, 25]. Other well known dataflow systems include Google Dataflow [2], IBM Streams [18], Apache Spark [66] and Spark Streaming [68], Timely Dataflow [52],

² <https://github.com/aversey/abscoq>

NebulaStream [69], Portals [60], and more [7, 56]. The popularity and wide-spread use of dataflow systems [25, 48] is due to their ability to scale-out production workloads. In particular, they provide high throughput, low latency, and strong guarantees (such as failure transparency, sometimes referred to as exactly-once processing). The programming model of most stateful dataflow systems is based on acyclic dataflow graphs [24]. In these graphs, the nodes are stateful processing tasks, and the edges are streams of data. As failure transparency is an important aspect of the stateful dataflow programming model, it and its failure recovery protocol is the focus of this paper.

2.3 Asynchronous Barrier Snapshotting

The failure recovery protocol used in Apache Flink [13] is a checkpointing-based rollback recovery protocol [21], in which the system regularly takes checkpoints and, after a failure, recovers to the latest completed checkpoint. For batch execution systems, such as MapReduce [19], the general approach is to atomically execute one batch at a time, and if a failure occurs, the system restarts from the beginning of the current batch. In contrast, computation on stateful dataflow streaming systems is continuous [24], without predefined recovery points in its execution, complicating the failure recovery. The solution to recovery in continuous computations is the acquisition of causally consistent snapshots [14], which can be used for the recovery to a consistent system state after a failure [21]. The specific implementation of Apache Flink [13] and other stateful dataflow systems [60] use the Asynchronous Barrier Snapshotting (ABS) protocol [12], an extended and optimized variant for data processing graphs of the Chandy-Lamport snapshotting protocol [14], for taking causally-consistent snapshots. In contrast to the Chandy-Lamport snapshotting protocol, the ABS protocol is tailored to acyclic dataflow graphs and its snapshots do not contain any in-flight events. In contrast to batching protocols, the ABS protocol is fully asynchronous, and does not require blocking coordination. For these reasons, the ABS protocol greatly benefits the end-to-end latency and throughput of the system.

2.4 Basic Notation

Functions. We denote a function f similarly to set-builder notation as: $[k \mapsto t \mid k \in \text{dom}(f)]$. The part after the bar defines the domain of the function. The part before the bar defines the value of the function at point k by the expression t . The expression t captures all variables defined on the right side of the bar, including k . A function with only one element in its domain is represented as $[x \mapsto x']$, for example, $[3 \mapsto 7]$ is such a function that $\text{dom}([3 \mapsto 7]) = \{3\}$ and $[3 \mapsto 7](3) = 7$. We denote function update as $f g$, such that:

$$(f g)(x) = \begin{cases} g(x) & \text{if } x \in \text{dom}(g) \\ f(x) & \text{if } x \notin \text{dom}(g) \end{cases}$$

Sequences. We represent a sequence S as a function f with domain $\{i \mid i \in \mathbb{N} \wedge i < |S|\}$. The length of the sequence is represented by $|S|$ and may be infinite. The notation S_i stands for the i -th element of the sequence S and equals $f(i)$. To simplify our analysis of sequences, we use $[t]_i^n$ as a shorthand for $[i \mapsto t \mid i \in \mathbb{N} \wedge i < n]$, where t is an expression that captures i and represents the i -th element of the sequence. Therefore, for any sequence S , we have that $S = [S_i]_i^{|S|}$.

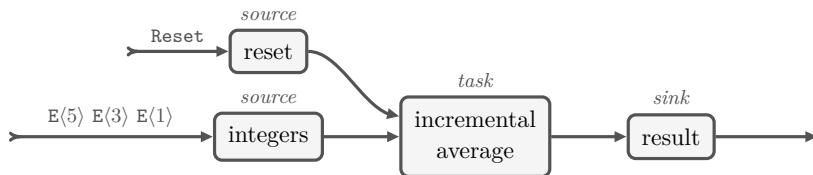


Figure 2 Example stateful dataflow program calculating the incremental average of a data stream of integers. Another stream is used to transfer control messages resetting the state of the program.

The usage of indices for variables standing for sequences may differ from other variables. If S stands for a sequence, then S_i corresponds to the i -th element of S . If, in contrast, x is not a sequence, then x_i is an independent variable and is not connected to x or any x_j . To avoid confusion, we name sets and sequences using uppercase and individual elements using lowercase.

Sequence concatenation can be used to extend or shrink existing sequences. We include a shorthand notation for sequence concatenation, concatenating S with S' as follows $S : S' \equiv [i \mapsto S_i \mid i \in \mathbb{N} \wedge i < |S|] [j + |S| \mapsto S'_j \mid j \in \mathbb{N} \wedge j < |S'|]$. To simplify extraction and addition of single elements, we denote single-element sequences $[x]_i^1$ as $[x]$, where x is the only value in the sequence. The empty sequence is represented as ε .

3 Stateful Dataflow

Stateful dataflow systems, sometimes also called *distributed dataflow*, *dataflow streaming*, or *stream processing* systems, are widely used for real-time processing of large amounts of streaming data. This section informally introduces the stateful dataflow programming model and its failure recovery mechanism, which we formalize and prove correct in later sections. It is mostly based on Apache Flink [13], a stateful dataflow system, however, the core concepts and techniques involved also apply to other similar systems [19, 66, 68, 2, 18, 52, 69, 60].

3.1 A Taste of Programming in Stateful Dataflow

Figure 2 shows a stateful dataflow example calculating the incremental average of a stream of integers. The example consists of two *sources* ingesting streams of events into the system. One source ingests a stream of integers $E\langle i \rangle$, and the other ingests a stream of `Reset` events. The term *stream* can be understood as an unbounded sequence of events, it may in general continue forever. The example also consists of a *task*, an internal processing unit, which calculates an incremental average of the integers. The incrementally computed averages are emitted to a *sink*, which is the output of the system.

A more detailed representation of the example is shown in Listing 1. Sources, tasks, and sinks are created using corresponding functions. The API enables users to: (1) create sources, tasks, and sinks; (2) specify the connections in the graph by providing input and output streams; and (3) to specify how the tasks process events by providing their processing functions. In this example, when the task receives an integer event $E\langle i \rangle$, it updates the average and emits the new average. When it receives a `Reset` event, it resets its local state, such that the average is reset to its initial state. To note is that the task is considered *stateful*, as it maintains local state for its computation of the incremental average, even though the processing function f is a pure function. Also to note is that it is possible to provide an easier-to-use API above this core API, for example an API based on higher-order functions (`map`, `flatMap`, etc.) [13, 60, 2, 52].

Listing 1 A stateful dataflow program calculating the incremental average of a data stream of integers (see Figure 2).

```
Source(input = "src_reset", output = "reset")
Source(input = "src_ints", output = "ints")
Sink(inputs = { "avgs" }, output = "sink_avgs")
Task(inputs = { "src_ints", "src_reset" }, output = "avgs",
  f = (event, state) => event match {
    case Reset =>
      val new_state = {sum = 0, count = 0}
      return (Nil, new_state)
    case E<value> =>
      val new_state = {sum = state.sum + value, count = state.count + 1}
      val average = E<value = new_state.sum / new_state.count>
      return (average : Nil, new_state) })
```

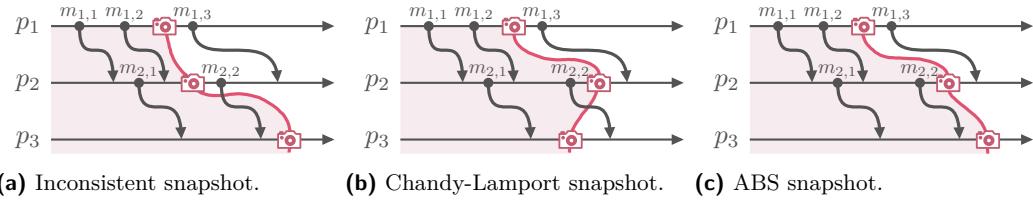


Figure 3 Examples of snapshots obtained in a distributed stateful dataflow system with three processes $p_1 \rightarrow p_2 \rightarrow p_3$.

3.2 Failure Recovery via Asynchronous Barrier Snapshotting

Failure recovery is a crucial aspect of stateful dataflow systems. In this section, we describe the failure recovery mechanism of the Asynchronous Barrier Snapshotting (ABS) protocol [12] as used in Apache Flink. More specifically, ABS is a *distributed snapshotting* protocol [14] which is used for the checkpointing-based rollback-recovery protocol [21] within Apache Flink [12]. After a failure, a checkpointing-based recovery will restart the system from the latest valid snapshot of the system [21].

Distributed Snapshotting Protocols. A distributed snapshotting protocol is considered *causally consistent* if it captures snapshots that do not violate causality [14]. Causality, here, refers to the causal order relation [35], informally: two events are causally ordered if one event was part of a causal chain leading to the other event. Consequently, a causally consistent snapshot captures the state of a system such that all events causally preceding any other event in the snapshot are included. This definition is illustrated by three example executions of different snapshotting protocols for a dataflow graph consisting of three nodes, shown in Figure 3. An incorrect implementation (Figure 3a) would be to let the processes periodically capture a snapshot of their state without coordination. A snapshot captured with this method can be inconsistent, thus not suitable for recovery, as it may violate causality. In the example, the incorrect snapshot has captured that $m_{2,2}$ was received by p_3 but never sent by p_2 , this is a violation of causality, and recovery from such a snapshot would be considered erroneous. In contrast, consistent snapshotting protocols do not violate causality. The Chandy-Lamport asynchronous snapshotting protocol [14] (Figure 3b) solves

■ **Listing 2** Representation of an event handler within a stateful dataflow system implementing failure recovery using the ABS protocol [12, 10].

```

EventHandler Def  $TK\langle f, [S_i]_i^n, o \rangle$ 
  Vars state, snapshots
  On Event Receive  $\langle S_j, \text{epoch}, \text{Event}\langle w \rangle \rangle$  If  $\exists v: \text{state} = \langle \text{epoch}, v \rangle$  Do
     $v', w' = f(v, w)$ 
    state =  $\langle \text{epoch}, v' \rangle$ 
    emit( $\langle o, \text{epoch}, \text{Event}\langle w' \rangle \rangle$ )
  On Event Receive  $\langle [S_i, \text{epoch}, \text{Border}]_i^n \rangle$  If  $\exists v: \text{state} = \langle \text{epoch}, v \rangle$  Do
    snapshots.update(epoch  $\mapsto$  v)
    state =  $\langle \text{epoch} + 1, v \rangle$ 
    emit( $\langle o, \text{epoch}, \text{Border} \rangle$ )
  On Event Fail Do
    state = Failed
  On Event Recover  $\langle \text{recoverEpoch} \rangle$  Do
    state =  $\langle \text{recoverEpoch}, \text{snapshots}(\text{recoverEpoch}) \rangle$ 

```

this issue through distributed coordination by means of disseminating markers during its regular execution, separating pre-snapshot and post-snapshot messages. However, a snapshot captured with the Chandy-Lamport protocol may capture in-flight events: as shown in the example (Figure 3b), the message $m_{2,2}$ was sent (according to p_2 's snapshot) but not yet received (according to p_3 's snapshot). The Asynchronous Barrier Snapshotting (ABS) protocol [12, 10], in contrast, captures complete distributed computations without in-flight events by modification of the marker-based Chandy-Lamport protocol. As shown in Figure 3c, the snapshot does not include any in-flight events.

The ABS Protocol. A representation of the ABS protocol [12, 10] corresponding to our formalization in Section 4 is found in Listing 2. The handler has two mutable states: the processing task's volatile **state**, and the persistent **snapshots** state. The **state** is a tuple $\langle \text{epoch}, v \rangle$ consisting of the current **epoch**'s sequence number being processed, and the state **v** of the processing task. The first event handler consumes an event **Event** $\langle w \rangle$ from a stream with stream name S_j out of the sequence of stream names S for some **epoch** if it is not currently in a failed state. It processes the event w on its current state v , which produces an output event w' and new state v' . It then updates its mutable state, and emits the output on its outgoing stream with stream name o . The second handler processes the **Border** markers from the higher-level ABS protocol. It will consume all border events from all its incoming streams in a single step. In doing so, it will take a snapshot of the local state and update the epoch number, as well as disseminate the border marker further downstream. To note is that the first handler does not consume from a stream if that stream has a border marker as its next event, instead it will block such streams until the border step (*i.e.*, the second handler) has been taken. The first and second handlers implement the ABS protocol, whereas the third and fourth handlers implement the failure recovery. The third handler models the random failures of tasks, a task can randomly fail at any time, in which case it loses its volatile state. The fourth handler implements the failure recovery, and is triggered by some external coordinating instance once it has detected the failure. Once a failure has

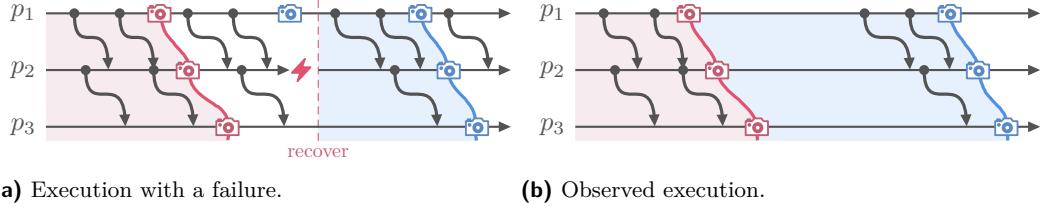


Figure 4 An execution with failures and its observed execution.

been detected, all tasks are recovered to the same epoch which corresponds to the latest snapshot of the system. When triggered, the fourth handler recovers the state back to the snapshot of the epoch found in the message.

Failure Recovery. The dataflow system can recover from failures using the ABS protocol. Figure 4a shows an execution using the ABS protocol in which \$p_2\$ fails. The coordinator (not displayed) will eventually discover the failure, and trigger a synchronous recovery step in which all processes recover to the latest completed snapshot and continue processing from there. Even though failures occur in the execution, the observer will be able to construct an idealized execution corresponding to our notion of failure transparency in which there are no failed events or incomplete epochs as shown in Figure 4b. This is, loosely speaking, achieved by ignoring the side effects from the failed epochs, and is explored in detail in Section 6.

4 Implementation Model

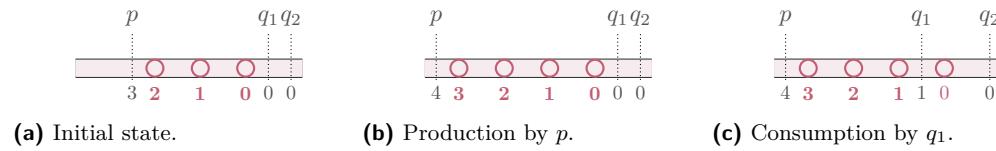
We now provide a formal model of the stateful dataflow system described above. The goal of this formalization is to capture and analyze key aspects of the implementation of the system, with focus on its failure recovery using the Asynchronous Barrier Snapshotting protocol [12, 10]. The formal model is presented in two parts: the first part presents an explicit evaluation rule for message passing, and the second part presents the evaluation rules for processing and failure recovery.

4.1 Streaming Model

The streaming model is based on processors (or tasks) that communicate via streams. A processor is a stateful entity that may consume an event from an incoming stream, process it, and produce events to its outgoing stream. Streams, in turn, transport the events between processors in a FIFO order. With this notion of processors and streams, we can execute computational graphs by means of steps. Note that, in this section, we discuss a general streaming model, leaving the implementation of processors abstract. Whereas, in the next section, we discuss concrete implementations of processors.

Syntax. Figure 5 shows the syntax of the streaming model. A configuration $c = \langle \Pi, \Sigma, N, M, D \rangle$ represents a point in an execution of a streaming program. The processors Π indexed by identifiers p represent processor definitions, for which Σ represents the states of the processors. The messages M are modeled as a sequence of all messages, for which a message m corresponds to a tuple of a sequence number n , a stream name s , and the message data d . The current sequence number from which a processor p reads from or writes to a stream s is represented by $N_p(s)$. The sequence numbers for all processors are represented by N . When

p, q	processor ID	s, o	stream name	$n \in \mathbb{N}$	sequence number
π	processor	σ	state	d	message data
$\Pi ::= [\pi]_p^{ \Pi }$	processors	$X ::= [x]_i^{ X }$			
$\Sigma ::= [\sigma]_p^{ \Pi }$	states	$x ::=$			action
$M ::= [m]_i^{ M }$	messages			$+ s d$	production
$N ::= [N_p]_p^{ \Pi }$	sequence numbers			$- s d$	consumption
$N_p ::= [s \mapsto n \mid s]$	sequence numbers of p			$m ::= n s d$	message

Figure 5 Streaming syntax.**Figure 6** Production and consumption to/from a stream with a producer p and consumers q_1 and q_2 .

a processor processes a message, it may produce and consume messages. This production and consumption is represented by a sequence of actions X . A production action producing message d to stream s has the form $+ s d$, similar to the consumption action $- s d$. The auxiliary data D is used to store global and additional execution information which is specific to the models; for example, it can be used to implicitly model the global coordinator. In the formalization here, the processor π , state σ , message data d and auxiliary global data D are seen as atomic values, that is, no information about their internal structure is provided. These limitations permit reusing the same syntax and rule for different instantiations of π , σ , d and D .

Figure 6 illustrates a stream as a sequence of messages with index numbers. When producing an event to a stream (Figure 6b), the event is appended to the stream with an incremented index number. This also increments the producer's index number for the stream from 3 to 4. Similarly, the consumer's index number points to the next event to be consumed. Figure 6c shows that the consumer q_1 has consumed the event 0, which in turn also increments its index number for the stream, pointing at the next event. Consumers and producers process the stream independently and asynchronously. The production of a message is a kind of broadcast, in the sense that all processors will have to consume it before consuming a newer message.

Step Rule. The streaming model essentially consists of a single rule (S-STEP) which describes the processing of messages. Intuitively, a streaming step from configuration (Π, Σ, N, M, D) can be taken if there is a local step with actions X , such that the actions are applicable. A local step describes how the processor Π_p changes its current state Σ_p to its next state Σ'_p using actions X . The actions X are applicable to N_p and M if all messages consumed by X are available on the input streams of the processor. The application of the actions X results in N'_p and M' , which are the incremented sequence numbers for the processor and the set of

messages M extended with the newly produced messages. In case of taking a streaming step, the configuration transitions to the new configuration $\langle \Pi, \Sigma[p \mapsto \Sigma'_p], N[p \mapsto N'_p], M', D \rangle$. In summary, the result of the streaming step is an update of the local state of the processor according to the local step, and an update of the sequence numbers and messages according to the actions X . To simplify the analysis of streaming steps, auxiliary information about the processor ID, its sequence numbers, and the actions of the step is placed on the arrow of the execution step. This information can be omitted when it is not needed by applying abstraction steps S-AbsX and S-ABSP.

$$\frac{\Pi_p \Vdash \Sigma_p \xrightarrow{X} \Sigma'_p \quad X(N_p, M) = (N'_p, M')}{\langle \Pi, \Sigma, N, M, D \rangle \xrightarrow[p]{N_p, X} \langle \Pi, \Sigma[p \mapsto \Sigma'_p], N[p \mapsto N'_p], M', D \rangle} \text{S-STEP}$$

$$\frac{c \xrightarrow[p]{N_p, X} c'}{\frac{c \Rightarrow c'}{c \Rightarrow c'}} \text{S-ABSX} \quad \frac{c \xrightarrow[p]{c \Rightarrow c'}}{c \Rightarrow c'} \text{S-ABSP}$$

The streaming rule can be applied if there exists a derivation of the form $\Pi_p \Vdash \Sigma_p \xrightarrow{X} \Sigma'_p$ for a processor Π_p . These are called local steps, since they have access only to the local data of a processor, *i.e.*, its definition, state and locally accessible messages. These rules describe the local step of a processor, in which the processor may produce and consume messages/actions X , and update its local state to Σ'_p . The produced actions X modify the sequence numbers of the processor N_p and the messages in the system after application. This is computed by the action application function $X(N_p, M)$ and results in the new sequence numbers N'_p and messages M' for the next configuration as defined below.

Action Application. The action application rule defines how actions modify the sequence numbers and messages. A production action $+ s d$ increases the sequence number of the stream s for the producer, and adds the message to the sequence of messages. Each stream has at most one producer; thus, we do not need to specify the producer in the action or message. A consumption action $- s d$ increases the sequence number of the stream s for the consumer, but does not remove it from the sequence of messages, as there may be other consumers waiting to consume the message. To note is that the consumption action application is only defined if the message is present in the sequence of messages. Due to this, local steps may only be applied in the context of the S-STEP rule if the consumed message is present in the sequence of messages. The remaining cases of the definition are for the recursive application of actions.

► **Definition 4.1** (Action Application).

$$(+ s d)(N_p, M) = (N_p[s \mapsto N_p(s) + 1], M \cup \{N_p(s) s d\})$$

$$(- s d)(N_p, M) = (N_p[s \mapsto N_p(s) + 1], M) \text{ if } N_p(s) s d \in M, \text{ undefined otherwise}$$

$$([x] : X)(N_p, M) = X(x(N_p, M))$$

$$\varepsilon(N_p, M) = (N_p, M)$$

According to the definition, it is not always possible to apply an action. This may be the case if, for example, a message for some sequence number is not yet available on its stream. This enables indirectly “passing” messages to the local step rules. Whereas the local step rule is defined for all possible steps for all messages that it may consume, cases in which the message consumption is not applicable by the action application definition are ruled out by the streaming global step rule. This leaves only messages which are applicable to be applied to the steps, thus passing the message to the rule.

v, w	value	$e \in \mathbb{N}$	epoch number
$\pi ::= \text{TK}\langle f, [S_i]_i^{ S }, o \rangle$	task	$d ::= \langle e, d_C \rangle$	message
$a ::= [e \mapsto v \mid e]$	snapshot archive	$d_C ::=$	message cases
$\sigma ::= \langle a, \sigma_V \rangle$	state	$\text{EV}\langle w \rangle$	event
$\sigma_V ::=$	volatile state	$\mid \text{BD}$	epoch border
$f1$	<i>failed state</i>	$D ::= M_0$	initial input messages
$\mid \langle e, v \rangle$	<i>normal state</i>		

Figure 7 Stateful dataflow syntax.

4.2 Stateful Dataflow Model

The presented stateful dataflow model consists of processing tasks, sources, and sinks. A processing task consumes messages from a set of input streams, and produces messages on its output stream. The task's behavior is defined by a function f which processes the messages. The function f takes the task's state and an input message, and produces a new state and a sequence of output messages: $f(v, w) = v', [W'_i]_i^n$. The presented formal model does not provide a syntax and semantics for functions; they can be expressed using any suitable formalism. The sources of the model are emulated by streams which are initialized in the first configuration to contain all the messages which are to be consumed from the source. That is, each source is represented by its output stream, which in turn becomes an input to one of the tasks of the computational graph. Sinks are also emulated as streams, however, in contrast to sources, they are initially empty. The computation of the system, informally, takes inputs from the sources, processes them in the processing graph, and produces outputs to the sinks.

Syntax. The syntax of the implementation model (Figure 7) extends the shared streaming syntax and semantics (Figure 5) by providing concrete instances of processors/tasks, messages, and state definitions. A task $\text{TK}\langle f, S, o \rangle$ is a three-tuple of its processing function f , sequence of input streams S , and its output stream o . Tasks process messages which are tuples of an epoch number e and the message data d_C . There are two kinds of messages: normal events $\text{EV}\langle w \rangle$ and epoch borders BD . The epoch border messages are markers used for the snapshotting algorithm, whereas the events are the actual data processed by the tasks. When processing, the tasks manipulate state which consists of a persistent *snapshot archive* a , *i.e.*, a map from epoch numbers to the corresponding local snapshots, and some *volatile state* σ_V . The snapshot archive is a map from epoch numbers e to the state v of the processor at the end of the epoch. The volatile state is either a *failed state* $f1$ or a *normal state* $\langle e, v \rangle$, consisting of the current epoch number and the state data value v of the processor. As with the messages, normal states are tagged by epoch numbers. A processor is in a failed state if it has crashed and lost its volatile state. The auxiliary data D used for this model consists of the initial input messages for the system. As we may need to restore the messages which are yet to be consumed, we keep track of all the initial input messages as the global auxiliary data of the system.

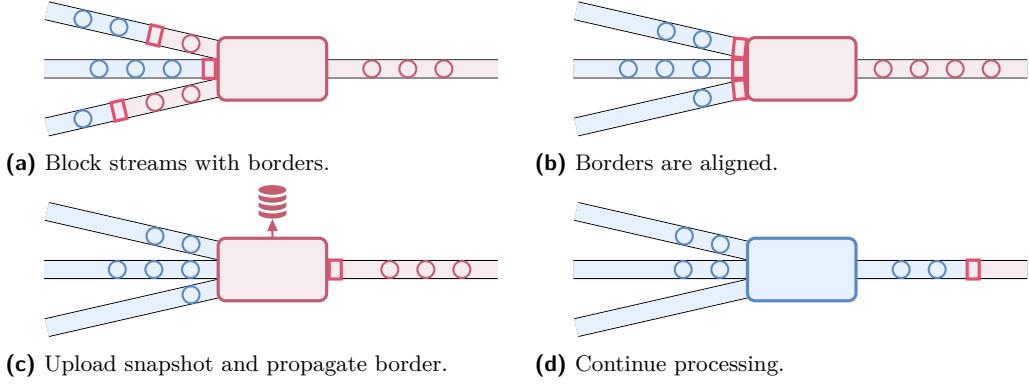


Figure 8 Epoch border alignment protocol (figure adapted from [10]).

4.2.1 Derivation Rules

The semantics of the model consists of seven rules. Three of the rules, I-EVENT, I-BORDER, and F-FAIL, are local rules which enable deriving a local step of the form $\pi \Vdash \sigma \xrightarrow{X} \sigma'$. Whereas the I-EVENT and I-BORDER rules model the processing of the system, the F-FAIL rule models nondeterministic crash-failures of a processing task within the system. These rules, together with the streaming rule S-STEP and its abstraction rules S-ABSX and S-ABSP, are used for deriving global steps. The fourth rule, F-RECOVER, is a global rule used for recovering the state of all processors after a failure.

Event Rule. The first rule, I-EVENT, models tasks processing events:

$$\frac{}{\text{TK}\langle f, S, o \rangle \Vdash \langle a, \langle e, v \rangle \rangle \xrightarrow{[-S_j \langle e, \text{EV}(w) \rangle] : [+o \langle e, \text{EV}(W'_i) \rangle]} \langle a, \langle e, v' \rangle \rangle} \text{I-EVENT}$$

The rule can perform a local step for a task $\text{TK}\langle f, [S_i]_i^{|S|}, o \rangle$, if the current state of the task is a normal state $\langle e, v \rangle$, and the task can consume an event $\text{EV}(w)$ from one of its inputs S_j . Applying a task's function f to its current state v and the consumed event w results in the task's next state v' and a sequence of output events $[W'_i]_i^n$. The rule updates the state of the task to the new state $\langle e, v' \rangle$ and produces the output events $[\text{EV}(W'_i)]_i^n$ on the output stream o . The local step produces the actions which are the concatenation of the consumed and produced events. For example, $[-S_j \langle e, \text{EV}(w) \rangle] : [+o \langle e, \text{EV}(w') \rangle]$ is the action of consuming the event $\text{EV}(w)$ with epoch number e from the input stream S_j and producing the event $\text{EV}(w')$ with epoch number e on the output stream o .

Border Rule. Whereas the event rule consumes a single event from a stream, the border rule (I-BORDER) consumes one border event BD from *every incoming stream*:

$$\frac{}{\text{TK}\langle f, [S_i]_i^n, o \rangle \Vdash \langle a, \langle e, v \rangle \rangle \xrightarrow{[-S_i \langle e, \text{BD} \rangle]_i^n : [+o \langle e, \text{BD} \rangle]} \langle a[e \mapsto v], \langle e + 1, v \rangle \rangle} \text{I-BORDER}$$

This consumption is enabled for a task if the next event to be consumed on every one of its incoming streams is a border event. In other words, the event rule consumes events up until all streams are aligned by the border events, at which point the border rule consumes

the border events from all its incoming streams. The rule is a local step which, in addition to consuming border events from all incoming streams and producing a border event on its outgoing stream, stores the current state v for epoch e to the snapshot storage a (by setting the new snapshot archive to $a[e \mapsto v]$), as well as incrementing the current epoch number.

Epochs are a key concept of Asynchronous Barrier Snapshotting. Each epoch is a sequence of data-bearing *events*, ending with an *epoch border*, and are used to define the boundaries of state snapshots. After regular processing for which some streams are blocked by border events (Figure 8a), the rule aligns the streams by the borders (Figure 8b), takes a copy of the current state of the processor storing it to the snapshot archive (Figure 8c), and propagates the epoch border message downstream and increments the epoch number, ready to process events from the next epoch (Figure 8d). The effect of this is that epochs of events are separated by the border events throughout the whole processing graph.

Failure Rule. Failures are introduced nondeterministically by the F-FAIL rule:

$$\frac{}{\text{TK} \langle f, S, o \rangle \Vdash \langle a, \sigma_V \rangle \rightarrow \langle a, \mathbf{f1} \rangle} \text{F-FAIL}$$

The failure rule sets the task's state to failed $\langle a, \mathbf{f1} \rangle$, thus losing the task's volatile state. Once a task is failed, it is no longer able to apply the steps I-EVENT and I-BORDER, and will remain idle until the F-RECOVER rule has been applied.

Failure Recovery Rule. The last rule, F-RECOVER, is a global rule which recovers the state of all failed tasks:

$$\frac{\langle a, \mathbf{f1} \rangle \in \Sigma}{\langle \Pi, \Sigma, N, M, M_0 \rangle \Rightarrow \text{lcs}(\langle \Pi, \Sigma, N, M, M_0 \rangle)} \text{F-RECOVER}$$

The rule may be triggered nondeterministically if there exists a task in a failed state, and will reset the state of the system to the latest common snapshot. The full details of how the latest common snapshot (lcs) is computed is discussed further below, as it depends on additional definitions.

The latest common snapshot is constructed by: (1) calculating the greatest common epoch for which a snapshot has been taken by all processors in the system; (2) restoring the state of all processors to their local snapshots at the greatest common epoch; and (3) restoring sequence numbers and messages to undo any messages that were produced or consumed for epochs greater than the greatest common epoch. The greatest common epoch is calculated by finding the minimum (*common*) of the maximum (*greatest*) epoch numbers of the local snapshots of all the processors.

► **Definition 4.2** (Greatest Common Epoch Number). *The greatest common epoch number of a configuration $c = \langle \Pi, \Sigma, N, M, D \rangle$ is:*

$$\text{gce}(c) = \min \{ \max(\text{dom}(a)) \mid \Sigma_p = \langle a, \sigma_V \rangle \}$$

The persistent *output messages* of the system consist of all messages produced up to and including the greatest common epoch. These messages can be identified by comparing their epoch number e to the greatest common epoch number $e \leq \text{gce}(c)$. The recovery purges any messages which are not part of this set, bar the initial input messages M_0 , thereby making these output messages (identified by out) persistent.

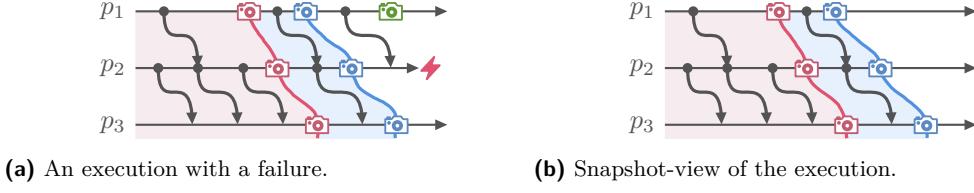


Figure 9 Executions viewed through the latest common snapshot.

► **Definition 4.3 (Output Messages).** For a configuration $c = \langle \Pi, \Sigma, N, M, D \rangle$, its output messages are:

$$\text{out}(c) = \{ns \langle e, d \rangle \mid (ns \langle e, d \rangle) \in M \wedge e \leq \text{gce}(c)\}$$

► **Definition 4.4 (Messages on a Stream).** The subset $M \downarrow s$ of messages on a particular stream is defined as:

$$M \downarrow s = \{n' s' d' \mid (n' s' d') \in M \wedge s' = s\}$$

The lcs function computes the latest common snapshot of a configuration for use as a recovery point in the F-RECOVER rule. Its computation makes use of the greatest common epoch number (gce), and the output messages (out). The states Σ' are restored by removing any stored snapshots with an epoch number larger than the gce, and the volatile states are restored to the states captured by the snapshot of the gce. The messages are updated to only keep the stable output messages $\text{out}(c)$ and the messages which are yet to be consumed M_{in} . The sequence numbers N' are updated accordingly, setting the sequence number of a processor p for a stream s to the number of messages that the processor has either produced or consumed on the stream: $|\text{out}(c) \downarrow s|$. Its complete definition is given below.

► **Definition 4.5 (Latest Common Snapshot).** The latest common snapshot of a configuration $c = \langle \Pi, \Sigma, N, M, M_0 \rangle$ is a configuration described by $\text{lcs}(c)$:

$$\text{lcs}(c) = \langle \Pi, \Sigma', N', M_0 \cup \text{out}(c), M_0 \rangle, \text{ where}$$

$$\begin{aligned} \Sigma' &= [p \mapsto \langle A(a), \langle \text{gce}(c) + 1, a(\text{gce}(c)) \rangle \rangle \mid \Sigma_p = \langle a, \sigma_V \rangle] \\ A(a) &= [e \mapsto a(e) \mid e \in \text{dom}(a) \wedge e \leq \text{gce}(c)] \\ N' &= [p \mapsto [s \mapsto |\text{out}(c) \downarrow s| \mid s \in \text{dom}(N_p)] \mid p \in \text{dom}(N)] \end{aligned}$$

Viewing computations through the lens of the latest common snapshot shows configurations which are caused by failure-free executions. Figure 9a shows an execution with a failed processor p_2 and an incompletely processed epoch (green). In contrast, the latest common snapshot view of the same execution (Figure 9b) shows only the two completed epochs (red, blue), masking the failed epoch. The snapshot is emulating an execution such that all the steps on epochs after the greatest common epoch are not taken, and all failed steps of incompletely processed epochs are ignored. This reasoning is further elaborated for the proof of failure transparency in the next section, where we show that the implementation model is failure transparent when viewed through the lens of the output messages function.

4.3 Assumptions

We make the following assumptions as a means to distill the essential mechanism of the failure recovery protocol. We assume that the message channels are FIFO ordered, a common assumption for snapshotting protocols [14]. With regard to failures, we make common

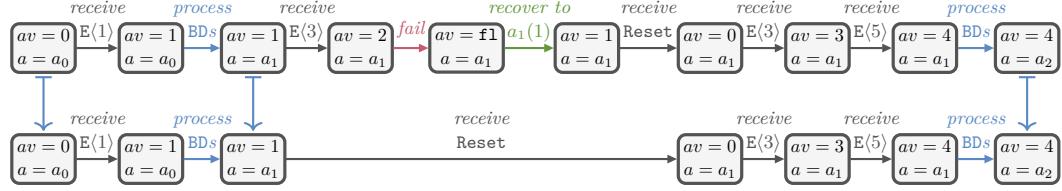


Figure 10 Execution of the incremental average task (Figure 2). Top: execution with a failure and subsequent recovery. Bottom: corresponding failure-free execution. Snapshot archives: $a_0 = [0 \mapsto 0]$, $a_1 = a_0[1 \mapsto 1]$, $a_2 = a_1[2 \mapsto 4]$.

assumptions to asynchronous distributed systems [9]. Failures are assumed to be crash-recovery failures, in which a node loses its volatile state from crashing. Further, we assume the existence of an eventually perfect failure detector, which is used for (eventually) triggering the recovery. With regard to system components, we assume the following components which can be found in production dataflow systems. The implicit coordinator instance is assumed to be failure free; in practice it is implemented using a distributed consensus protocol such as Paxos [37]. The snapshot storage is assumed to be persistent and durable; a system such as HDFS [57] would provide this. Further, the input to the dataflow graph is assumed to be logged such that it can be replayed upon failure. In practice, a durable log system such as Kafka [32] would be used for this. For our model, we make the following assumptions. The recovery is assumed to be an atomic, synchronous system-wide step. In practice, it may be implemented as an asynchronous atomic step, which allows tasks to start processing before all have been recovered. Further, the task's processing functions are assumed to be pure, *i.e.*, free from side effects. A function f may be re-executed multiple times due to failures; a common assumption in related work [8, 30].

5 Failure Transparency

In this section, we define failure transparency such that it can be applied to systems described in small-step operational semantics with distinct failure-related rules. We first provide a rationale behind failure transparency, followed by its formalization.

5.1 Rationale

The purpose of failure transparency is to provide an abstraction of a system which hides the internals of failures and failure recovery. In particular, we would like to be able to show that the implementation model presented in the previous section is failure transparent. In concrete terms, this entails showing that executions in the implementation model can be “explained” by failure-free executions, something which we explore in this section.

Consider the task of computing the incremental average from the previous example (Section 3, Figure 2). The task consumes regular events $E\langle i \rangle$, reset events, and border events BD. For this example, we consider a partial execution of the task in which it processes the events: $[E\langle 1 \rangle, BD, E\langle 3 \rangle, \text{fail}, \text{recover}, \text{Reset}, E\langle 3 \rangle, E\langle 5 \rangle, BD, \dots]$. The task's configurations consist of the task's current average value av , and its snapshot archive, a . Figure 10 shows at the top an execution of the task with a failure and subsequent failure recovery as the fourth and fifth events. After the recovery step, in its sixth configuration, the task's state is reset to its state for the snapshot $a_1(1)$, at which point it had the average value 1.

The question we ask is whether we can rely on the behavior of the task? More specifically, can we use the average value $av = 2$ in the fourth configuration (after receiving the event $E(3)$)? The problem is that the task will fail in its next step, and recover to a state in which the receiving of the event has been undone. Moreover, the task continues its execution after recovery by processing the reset event first, and does never reach a state again in which its average value is 2. For this reason, we cannot blindly rely on the observed behaviors of the task as we may observe things which are later undone. In more complex systems, failures may further result in duplications and reorderings of events, further complicating the reasoning about the system.

Dealing with these issues requires the observer of the system to reason about which events are effectful and which are to be discarded. In some sense, the observer should be able to reason about the observed execution as if it was an ideal, failure-free execution, *i.e.*, an execution in which all events are effectful. Put in another way, the solution is to find a corresponding failure-free execution, and reason about that one instead. Intuitively, the observer should find some failure-free execution which “explains” the execution. Considering the above example, a failure-free execution thereof would correspond to the bottom execution in Figure 10. Note that there are no failure or recovery steps in the failure-free execution, yet its state progresses in a similar way to the original execution.

Even though the failure-free execution on an intuitive level correspond to the original execution, we would like to have a formal notion for this. The idea is to lift the observed executions by means of “observability functions”, to a level where failure-related events and states are hidden. For example, for the executions above, we could define an observability function which takes the configuration of the task and keeps only the snapshot storage. After this transformation, applying this function to every configuration in the executions, we will not be able to distinguish the two executions by observing the system at any point in time. That is, common to both executions, we will first observe a_0 , then a_1 , and finally a_2 . On a technical level, for every configuration of the original execution, we can find a configuration in the failure-free execution which, after application of the observability functions, is equal to it (*e.g.*, the mapping from top to bottom configurations in Figure 10); this is what we mean by “observable explainability”. Thus, we can explain the original execution by the failure-free execution using the provided observability function.

The essence of our definition of failure transparency is derived from the notion of explaining the original executions by failure-free executions using observability functions. Instead of reasoning about executions, we can reason about the observable output of executions at any given moment. Using observability functions effectively hides the internals of the model and enables the user to focus on the output of the system. That is, the user can reason about failure-free executions instead of faulty executions.

This informal introduction highlights three essential parts of failure transparency: the execution system, failures within the system, and the observability of the system. The goal of the rest of this section is to define these terms and to provide a formal definition of *failure transparency*.

5.2 Executions

The execution system for the failure transparency analysis is modelled as a transition system for which the transition relation is provided as a set of inference rules. In particular, we provide a formal definition for executions as a means to discuss the execution of systems. With this notion, distributed programs can be formally modelled in small-step operational

semantics, and consequently formally verified. Although it may seem unintuitive to model distributed systems as transition systems for which the transition relation is defined over the global state, this is in fact commonly done in other formal frameworks such as TLA⁺ [38].

► **Definition 5.1** (Execution Step). *A statement $c \Rightarrow c'$ is called an execution step from c to c' . We denote the derivability of an execution step in the set of rules R by $R \vdash c \Rightarrow c'$.*

We reason about systems in terms of their executions. An execution is a sequence of configurations C , connected by execution steps derivable in a set of rules R , starting from some initial configuration C_0 .

► **Definition 5.2** (Executions). *A sequence of configurations $[C_i]_i^n$ is called an execution in a set of rules R , if $\forall i < n. R \vdash C_{i-1} \Rightarrow C_i$. The set of all possible executions starting from C_0 in R is denoted as $\mathbb{E}_{C_0}^R$.*

The set of rules R of an execution specifies its reducibility relation by providing $c \Rightarrow c'$ as a conclusion of some of its rules. This approach is commonly known as *small-step operational semantics*. In our representation, the set of rules is explicit, whereas commonly it is implicit. This is due to our need to explicitly distinguish between separate execution systems. This allows us, for example, to separate an execution system into two parts: one with failures R s.t. the failure-related rules are a subset thereof $F \subseteq R$, and one without failures $(R \setminus F)$.

5.3 Observational Explainability

The observability function represents the observer's view of the system. It notably differs from the plain configurations in the following two ways: the observer may not observe all internal details of configurations, *i.e.*, some parts of the configuration are *hidden* from the observer (*e.g.*, hiding commit messages [8]); and the observer may observe some derived views of the configuration.

► **Definition 5.3** (Observability Function). *An observability function O of an execution system is a function which maps configurations to their observable outputs. It is required to be monotonic with respect to execution steps possible in the set of rules R for some partial order \sqsubseteq_O , that is: $\forall c, c'. (R \vdash c \Rightarrow c') \implies O(c) \sqsubseteq_O O(c')$.*

We say that an implementation's execution is observably explained by a specification's execution, if the observer cannot distinguish the two executions. This is the case when, for every configuration in the implementation's execution, there is a corresponding configuration in the specification's execution, such that their observed values are equal after application of the respective observability functions.

► **Definition 5.4** (Observational Explanation). *A sequence of configurations C of length n is explained by a sequence of configurations C' of length n' with respect to observability functions O and O' , denoted as $C \stackrel{O}{\rightleftharpoons} \stackrel{O'}{=} C'$, if:*

$$\forall m < n. \exists m' < n'. O(C_m) = O'(C'_{m'})$$

An implementation's system, in turn, is observably explainable by the specification's system, if for each execution of the implementation there exists an explaining execution in the specification. We call this property *observational explainability*.

► **Definition 5.5** (Observational Explainability). *The set of rules R is observationally explainable by R' with respect to their observability functions O and O' and the translation relation T , denoted as $R \stackrel{O \xrightarrow{T} O'}{\rightleftharpoons} R'$, if:*

$$\forall c' \in \text{dom}(T). \forall c. c' T c \implies \forall C \in \mathbb{E}_c^R. \exists C' \in \mathbb{E}_{c'}^{R'}. C \stackrel{O}{\rightleftharpoons} \stackrel{O'}{=} C'$$

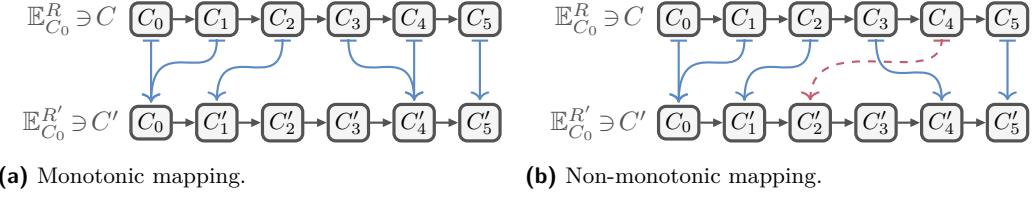


Figure 11 Monotonic and non-monotonic mapping of configurations.

Properties of Observational Explainability. Observability functions are required to be monotonic, since observations should be regarded as stable. That is, once a value has been observed, then it should remain observable in the future. The system should not be able to undo something that has been observed, otherwise the observer would not be able to rely on the output. The reason for this is twofold. First, an observer may observe the system multiple times, and newer observations should provide more up-to-date views. Second, the sequence of observations should correspond to a valid explanation with respect to the higher-level specification, this is explored next.

In the general case, it is desirable to have a monotonic mapping of configurations between the abstract-level and implementation-level executions. Figure 11a shows a monotonic mapping of configurations between an implementation (top) and a specification (bottom). What makes the mapping monotonic is that each subsequently mapped configuration of the implementation is mapped to a configuration with a monotonically growing index. Figure 11b, on the other hand, shows a non-monotonic mapping, as indicated by the red dashed line. Non-monotonic mappings, however, are not considered valid explanations. For example, if the specification consists of the sequence a followed by b , then an implementation which produces b followed by a is not considered a valid implementation thereof. Thus, we should not use non-monotonic mappings for the explainability of executions. We capture this notion in the definition of monotonic observational explanation.

► **Definition 5.6** (Monotonic Observational Explanation). *An observational explanation is monotonic if it is a monotonic mapping of configurations. That is, $[C_i]_i^n$ is monotonically explained by $[C'_j]_j^{n'}$ w.r.t. O and O' if:*

$$\exists [h_k]_k^n. (\forall k < n. \forall k' \leq k. h_{k'} \leq h_k) \wedge (\forall m < n. \exists m' = h_m < n'. O(C_m) = O'(C'_{m'}))$$

The following lemma explicitly shows that our definition of *observational explainability* is equivalent to the definition of *monotonic observational explainability*. That is, our definition does not have the problem with non-monotonic mappings of configurations since the observability functions are required to be monotonic. For this reason, we do not distinguish between the two definitions in the following sections.

► **Lemma 5.7.** *If R is observationally explainable by R' w.r.t. O , O' , T , then it is also monotonically observationally explainable:*

$$\forall c' \in \text{dom}(T). \forall c. c'Tc \implies \forall C \in \mathbb{E}_c^R. \exists C' \in \mathbb{E}_{c'}^{R'}.$$

C is monotonically explained by C' w.r.t. O and O'

Proof. The complete proof is available in the companion technical report [62]. ◀

To further aid the use of these definitions within proofs, we also show that the definition of observational explainability is transitive, as well as a compositionality lemma on the observability functions. The parametrization of the observable explainability enables reasoning

about models which differ in their initial states, and for which we want to apply different observability functions at the different levels. That is, it can be used for reasoning about sets of rules which differ in their initial states, and for which we want to apply different observability functions at the different levels.

► **Lemma 5.8** (Transitivity). $R \xrightarrow{O} R' \wedge R' \xrightarrow{O'} R'' \implies R \xrightarrow{O \circ O'} R''$

Proof. The complete proof is available in the companion technical report [62]. ◀

► **Lemma 5.9** (Composition). $\forall O''. R \xrightarrow{O} R' \implies R \xrightarrow{O'' \circ O} R'$

Proof. The complete proof is available in the companion technical report [62]. ◀

5.4 Defining Failure Transparency

The general goal of failure transparency is to provide an abstraction of a system which masks failures from the users. We express this notion using observational explainability between the implementation and its failure-free part. That is, the implementation should be observationally explainable by the implementation without failures. By explicitly separating the set of failure-related rules F , it is easy to define the two systems: namely, the implementation system with all rules, *i.e.*, R ; and another system with all rules except the failure-related rules, *i.e.*, $R \setminus F$. To fully instantiate the observational equivalence, we further use the same observability function O on both the low and high levels, and as a translation relation we use the identity relation on the set of initial configurations.

► **Definition 5.10** (Failure Transparency). *A set of rules R is failure-transparent with respect to failure rules $F \subseteq R$ for a monotonic observability function O and a set of initial configurations K , this is denoted as $R \mathbin{\Downarrow}_K^O F$, iff:*

$$R \xrightarrow{O \mathbin{\Downarrow}_{\{(c, c) \mid c \in K\}}^O} O(R \setminus F)$$

6 Failure Transparency of Stateful Dataflow

In this section, we show that the presented implementation model (Section 4) is failure transparent (Definition 5.10) for the observability function out (Definition 4.3). In order to prove this, instead of reasoning about executions directly, we reason about the traces of steps which are performed to obtain these executions. This simplifies the proof, enabling us to reorder and remove specific steps in and from a trace; in contrast, doing the same with a configuration from an execution affects all following configurations. In this section, we first define traces and a causal order relation on traces, and then prove the failure transparency of the implementation model by manipulating traces. Finally, we complete our analysis of the model by formulating and proving its liveness, showing that the implementation model eventually produces outputs for all epochs in its input.

6.1 Traces and Causality

A trace is a sequence of steps, for which each step is a compact representation of the derivation of a transition from one configuration to another.

► **Definition 6.1** (Trace). *A trace Z is a sequence of trace steps. A trace step z is one of: $\langle \text{I-EVENT}, p, N_p, X \rangle$; $\langle \text{I-BORDER}, p, N_p, X \rangle$; $\langle \text{F-FAIL}, p \rangle$; $\langle \text{F-RECOVER} \rangle$. Here I-EVENT, I-BORDER, F-FAIL, and F-RECOVER play the role of the discriminant, where the trace step is a tagged union.*

For example, if in the derivation tree of an execution step from the i th to the $i + 1$ th configuration, i.e., of $R \vdash C_i \Rightarrow C_{i+1}$, F-RECOVER was the root rule, then this execution step corresponds to the step $\langle \text{F-RECOVER} \rangle$ in the trace. To link traces with executions, we use the following definition of trace application.

► **Definition 6.2** (Trace Application). *A trace Z of length n applied to a configuration c results in a sequence of configurations C of length $n + 1$, i.e., $Z(c) = C$, if, for all steps Z_i , the represented derivation of an execution step can be applied to the i th configuration producing the $i + 1$ th configuration.*

Traces can be generated from executions; however, not every trace corresponds to an execution. This may be the case if a trace has been constructed incorrectly, or reordered in some way. For this reason, we define valid traces, which are traces that correspond to executions.

► **Definition 6.3** (Valid Trace). *A trace Z is valid from configuration c if it is applicable to it, i.e., if there exists an execution $C \in \mathbb{E}_c^I$ such that $Z(c) = C$.*

As the proof reasons about the reordering of steps in a trace, it is important to formulate which reorderings of steps preserve the validity of the trace. To handle this, we define a causal order relation on trace steps similar to the happens-before relation [35], and show how it can be used to reason about traces.

► **Definition 6.4** (Causal Order). *(See technical report [62] for the formal definition) A step Z_i happens before Z_j with $i < j$ if:*

1. *One of them is an F-RECOVER step (global recovery)*
2. *They both occur on the same processor (intraprocessor order)*
3. *If Z_i produced a message which is consumed by Z_j (interprocessor order)*
4. *If there exists some step Z_k such that Z_i happens before Z_k and Z_k happens before Z_j (transitivity)*

Finally, we state a lemma that causality-preserving permutations, i.e., permutations that preserve the causal order relation [62, Definition B.5], also preserve the validity and the end result of their application. Intuitively, it follows from the fact that causally unrelated steps should not influence each other.

► **Lemma 6.5** (Application of Causality-Preserving Permutations). *For a trace Z valid from c with size $|Z| = n$, if Z' is a causality-preserving permutation of Z , then: Z' is valid from c ; Z and Z' end in the same configuration after application to c , i.e., $Z(c)_n = Z'(c)_n$.*

Proof. The complete proof is available in the companion technical report [62]. ◀

6.2 Proving Failure Transparency

As it is required by the definition of failure transparency, we first define the sets of rules, namely I, F, and $(I \setminus F)$; and the set of valid initial configurations K .

The semantics of the model consist of seven rules, defining two separate sets of rules. The set of rules with failures I consists of all seven rules that have been defined for the stateful dataflow implementation model; it corresponds to the implementation model presented in

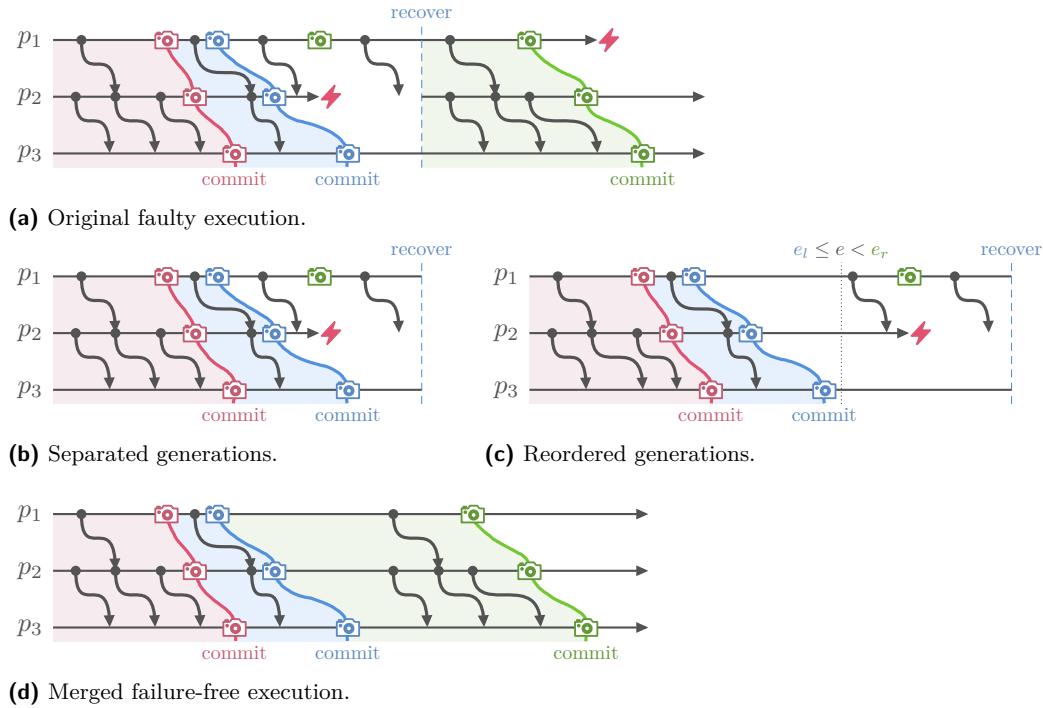


Figure 12 The step-wise construction of a failure-free execution trace from an execution with failures.

Section 4. The set of failure-related rules F within the implementation model consists of the two rules F-FAIL and F-RECOVER. This way, the rules without failures are defined as the set $(I \setminus F)$.

► **Definition 6.6** (Implementation Model Rules). $I = \{S\text{-STEP}, S\text{-ABSX}, S\text{-ABSP}, I\text{-EVENT}, I\text{-BORDER}\} \cup F$

► **Definition 6.7** (Failure-Related Rules). $F = \{F\text{-FAIL}, F\text{-RECOVER}\}$

The sets of initial configurations which are considered are any acyclic graph structures which are properly initialized.

► **Definition 6.8** (Valid Initial Configurations). $K = \langle \Pi, \Sigma, N, M, M_0 \rangle$ such that: the graph defined by Π is acyclic, and the tasks' functions f do not output infinite sequences; Σ are the initial well-formed states; N are sequence numbers initialized to 0 for the streams; M consists of the well-formed inputs to the streams; $M_0 = M$.

► **Theorem 6.9** (Failure Transparency of the Implementation Model). $I \setminus K^{\text{out}} F$, i.e., the set of rules $I = \{S\text{-STEP}, S\text{-ABSX}, S\text{-ABSP}, I\text{-EVENT}, I\text{-BORDER}\} \cup F$ is failure transparent with respect to the failure rules $F = \{F\text{-FAIL}, F\text{-RECOVER}\}$ for the observability function out and the set of initial configurations K .

Before proceeding with the proof itself, we provide a sketch of it. The proof idea is to construct a failure-free observational explanation of an arbitrary execution in the implementation model.

The construction is done using traces; we reorder and manipulate the original trace so that failures, recoveries, and discarded trace steps are removed from it. Figure 12 illustrates the construction: (1) first, we split the trace by the recovery steps into generations; (2)

next, the trace steps are reordered such that all discarded steps are moved to the end of the generation; (3) then, these steps are safely discarded; (4) finally, we concatenate the generations to get the final trace.

Next, we have to show that: (i) the constructed trace is valid, *i.e.*, it corresponds to a failure-free execution; and (ii) that the execution is an observational explanation of the original execution. We do so by reasoning about the preservation of validity and observable outputs in each step of the construction. For trace validity, the most complicated step is the reordering (step 2 of the construction). We show that the reordering is causality-preserving and thus, by Lemma 6.5, it produces a valid trace. For observational explanation, throughout the construction we maintain a mapping of observations from the steps of the original trace to the steps of the constructed trace. The challenge lies in the reordering of steps (step 2 of the construction) and the fusion of generations (step 4 of the construction). For the reordering, we show a lemma that the observable output is not changed by the discarded steps; and, for the fusion, we show that the latest common snapshot of a generation is exactly the configuration obtained by the reordering and removal of the discarded steps. This, accompanied by an analysis of the rules, lets us show that the sequence of observable outputs is the same for the original and the failure-free traces.

Proof. Expanding the definitions, we need to prove that, for all executions in \mathcal{I} with potential failures, there is an observational explanation in the failure-free model $(\mathcal{I} \setminus \mathcal{F})$. Given an arbitrary execution C of length n in \mathcal{I} from initial configuration $c \in K$, *i.e.*, $[C_i]_i^n \in \mathbb{E}_c^{\mathcal{I}}$, the goal is to construct a failure-free execution $[C'_j]_j^{n'}$ such that:

$$[C'_j]_j^{n'} \in \mathbb{E}_c^{\mathcal{I} \setminus \mathcal{F}} \wedge \forall m < n. \exists m' < n'. \text{out}(C_m) = \text{out}(C'_{m'})$$

This execution is constructed indirectly, by first constructing a trace Z' which then generates it. First, we need to prove that the constructed failure-free trace Z' is valid from c , *i.e.*, $Z(c)$; next, we need to show that the corresponding execution $C' = Z'(c)$ is an observational explanation of the original execution, that is, for each configuration in the original execution, we have to provide an observationally equal configuration in the constructed execution. From the original trace Z , for which $Z(c) = C$, we construct the failure-free trace Z' in four steps as outlined in the proof sketch and illustrated in Figure 12.

(1) First, the trace is split by the recovery steps into generations, giving us a sequence of generations G (Figure 12b). Each generation is a sequence of S-STEPs ending with an F-RECOVER step; in the case of the last generation it may not necessarily end with an F-RECOVER step. By construction, each generation is a valid trace as each of them is a contiguous part of a valid trace. We construct the observability mapping by mapping the configurations of the original trace to their closest preceding committing border steps. A *committing border step* is an I-BORDER step which changes the greatest common epoch number, gce, and thus also the observed output, out; such steps are labeled with “commit” in Figure 12. The equality of observations holds, since, by inspection of the rules, only a committing border step can change the observable output [62, Lemma B.6].

(2) Next, from each generation $g = G_i$, we construct a new reordered trace $g' = G'_i$ so that all the steps of epochs above the greatest common epoch of the generation are placed after the steps of epochs below it (Figure 12c). In effect, this moves all the discarded steps to the end of the generation, since they are discarded by the recovery, which in turn is done to the greatest common epoch of the generation. In other words, $g' = \text{filter}(x \in g. \text{epoch}(x) \leq e) : \text{filter}(x \in g. \text{epoch}(x) > e)$, where $e = \text{gce}(g|_{|g|-1})$ is the epoch number to which the recovery is done. The new traces are still valid, as the reordering is causality preserving [62,

Lemma B.7], and thus the validity follows from Lemma 6.5. The mapping of observations is kept intact, since the outputs of the committing border steps are not changed by the reordering [62, Lemma B.8]. This follows from the fact that the observable output is only changed by committing border steps [62, Lemma B.6], that causality-preserving permutations result in the same configuration (Lemma 6.5), and that the reordering is preserving causality.

(3) Then, from each G'_i we construct a new trace G''_i by removing the discarded steps and the recovery step. That is, the suffix consisting of the failure steps, recovery steps, and any steps of epochs greater than the greatest common epoch of the generation are removed. The new trace is a prefix of G'_i , and is thus still a valid trace [62, Lemma B.2]. We keep the same mapping of observations for the steps that were not removed. As, within a generation, only the suffix is removed, it does not affect the observed outputs of the remaining steps, and thus the mapping of observations is kept unchanged.

(4) Finally, we concatenate all stripped generations G''_i to get the merged trace Z' (Figure 12d). We show that the last configuration of each of the generations G''_i is exactly the latest common snapshot of the original generation G_i [62, Lemmas B.10-11], in other words, the latest common snapshot is a view of a configuration as if only the committed steps occurred. Since the recovery is done to the latest common snapshot, it is also the same configuration as the first configuration of the following generation G''_{i+1} . For this reason, the concatenation of all generations forms a trace Z' valid from c . The observed outputs are not changed by the merge, and we maintain the same mapping.

By these four steps we have constructed a failure-free observational explanation of the faulty execution, which means that the implementation model is observationally explainable (Definition 5.5) by its failure-free version, or, in other words, it is failure transparent (Definition 5.10). \blacktriangleleft

6.3 Liveness

The proposed definition of failure transparency is a safety property [3, 34], *i.e.*, it prohibits the implementation from reaching invalid states. Being as such, failure transparency does not require the implementation to take any observable execution steps; an implementation that never takes a step would trivially satisfy the property. In contrast, ensuring that the implementation eventually does something is a *liveness* property [3, 34]. To complete our analysis, we would like to show that the implementation model eventually produces outputs for all epochs in its input. This is a liveness property which, consequently, does not concern itself with the correctness of the outputs. However, in combination with the failure transparency property, the properties ensure that the presented implementation model *eventually* produces the *correct* outputs. For this reason, we prove the following theorem about the liveness of the implementation model.

► **Theorem 6.10** (Liveness of the Implementation Model). *For every input epoch present in the initial configuration, eventually a corresponding epoch appears in the output of a fair execution. That is:*

$$\begin{aligned} \forall k = \langle \Pi, \Sigma, M, N, D \rangle \in K. \forall C \in \mathbb{E}_k^I. \text{fair}(C) \implies \\ \forall (n s \langle e, d \rangle) \in M. \exists c \in C. \exists (n' s' \langle e', d' \rangle) \in \text{out}(c). e = e' \end{aligned}$$

where a fair execution is maximal (*i.e.*, it is not a prefix of another execution), has a finite amount of failures, and eventually executes any step which is eventually always enabled (see the technical report [62] for the formal definition of fair execution). \square

The liveness theorem states that for any fair execution C starting from a valid initial configuration k , and for all input epochs e , eventually there is a configuration c in the execution for which the output $\text{out}(c)$ contains the epoch e .

Proof. The complete proof is available in the companion technical report [62], it is summarized as follows. First, we show that it suffices to demonstrate that, continuing from any configuration c reachable from the valid initial configuration k , one or both of the following are true: eventually there is a failure; or eventually the epoch is visible in the output. As the considered executions have only finite amounts of failures, we further simplify the proof goal: it suffices to show that eventually the epoch appears in the output under the assumption that there are no more failures. We handle this simplified case by inductive reasoning on the acyclic dataflow graph of processors. The induction’s base case is the graph consisting of the source input streams but with no processors. The induction hypothesis states that all streams are well-formed and that the border message of all input epochs eventually appear on all streams; this is satisfied for the base case by validity of the initial configuration. Then, in the induction step, we construct the graph by adding one processor at a time, given that all of its input streams are already handled, as either source inputs or as outputs of other processors in the previous step’s graph. The assumption of fair scheduling allows us to reason about the processor locally, since, by definition of fairness, if a message has arrived to the processor, it will eventually be consumed. As a conclusion of the induction, each processor will eventually have processed a border of each epoch present in the initial configuration; thus, eventually all processors will process a border of each initial epoch. This, in turn, by analysis of I-BORDER , gce , and out , shows that the border messages of the epoch will eventually be in the output. ◀

7 Related Work

Failure Transparency, Observational Explainability. There has been a significant body of research on failure transparency [40]. To our knowledge, the earliest work on failure transparency was by von Neumann in 1956 [63] on creating reliable systems from unreliable components. Later work by Wensley in 1972 [65] discussed software techniques for failure transparent computing. Lowell and Chen discussed failure transparency in the context of consistent failure recovery protocols [44]. In their work, they introduced “equivalence functions” for comparing executions, a concept which inspired the observability functions in this paper. Our work, in contrast, restricts these functions to be monotonic, and discusses their application to both levels (low and high) of the system, which facilitates the presented transitivity lemma (Lemma 5.8). Around the same time as Lowell and Chen, Gärtner discussed general models for fault-tolerant computing [26]. Similar to our work, Gärtner separated fault-tolerant programs into two separate sets of rules (actions): the rules for normal behavior; and the rules for failure behavior. With this separation, Gärtner discussed various properties and forms of fault-tolerant programs. In the context of Gärtner’s work, our definition of failure transparency would be considered “failure masking”, in the sense that the system can recover from failures and continue its normal operation. Whereas these works defined failure transparency as a conjunct of *safety and liveness* [34, 3], we have only considered its safety property for our definition.

The presented definition for observational explainability is closely related to previous definitions of refinement (*e.g.*, TLA [38, 36], Compiler Correctness [53]), implementation (*e.g.*, I/O Automata [45]), and simulation. In simplified terms, one set of executions implements another if it is a subset thereof (modulo stuttering and multistep executions).

Our definition of observational explainability, in some sense, extends the notion of refinement to directly include a refinement mapping [1] on both sides via observability functions. It resembles notions from related work such as observational equivalence [8] and observational refinement [30]; in contrast to these works, we provide a formal definition thereof. Different from inductive proof approaches as typical for TLA [38] and simulation proof strategies, our proof approach reasons about the whole sequence. This makes it not necessary to include notions for ghost variables [49] (also known as auxiliary variables [39]) for the purpose of reasoning about past or future events.

Failure Transparency Proofs. Failure transparency and observational explainability can be proven in various ways. For example, Burckhardt et al. [8] prove “observational equivalence” for their serverless programming model. Mukherjee et al. [51] propose a failure transparency theorem for their system of reliable state machines: an execution of the implementation is a refinement of an execution without failures “with respect to its observable behavior”, reminiscent of our definition of failure transparency. Other works include models for distributed reliable actor communication [61], serverless microservices and observational refinement [30], and reliable state machines [51]. Their specific approaches may differ, some use simulation [8, 30], others model failures explicitly [30, 61, 51], and others use notions similar to observability functions [8]. Another approach is to prove the proper restoration of applications to the exact configuration as before the crash [50]. Our presented failure transparency proof shares similarities to the proof of the Asynchronous Barrier Snapshotting protocol [10], such as reasoning about causal orderings; however, our proof relies to a greater degree on abstraction in terms of refinement of models.

Distributed, Resilient Programming Models. Stateful dataflow has had a high impact [24] through systems such as: MapReduce [19], Apache Spark [67, 66], Apache Flink [12], Google Dataflow [2], IBM Streams [18], Portals [60], and others [7, 56]. However, there are other notable resilient programming models and systems, including: Pregel, a graph-based system [47], Resilient X10 [17], virtually resilient immortals [27], fault-tolerant reactives [50], thread-safe reactive programming [20], Durable Functions [8], stateful entities [54], the eXchange Calculus [6], and others [61, 30, 51, 15]. In general, these resilient programming models provide system means to recover from failures, the user does not need to implement the failure recovery mechanisms themselves. Actor models, in contrast, provide the users with manual failure-handling constructs. For example, the failure-handling constructs in Erlang, such as actor monitors and supervision [5], have been used successfully for building reliable services within the telecom industry [4]. Moreover, other programming models such as Argus [42] and transactors [22] provide constructs for transactions, which in turn can be used for building reliable services.

The formalization of distributed systems has been a long-standing research topic. Notably, formalization frameworks such as TLA [38] and I/O Automata [45], have been used to reason about distributed systems. Examples of this include a dataflow system that was formalized using I/O Automata [46]. The ABS protocol for stateful dataflow has been formalized with transition systems [10]. Recently, operational semantics have been used to model and reason about such systems [8, 61, 30, 51, 28].

Failure Recovery. A general overview of rollback-recovery protocols was given by Elnozahy et al. [21], comparing between checkpointing-based and logging-based protocols. Stateful dataflow systems use either checkpointing, or a combination of the two [7, 56, 2, 12, 64, 66,

19, 18]. The MapReduce system performs failure recovery by detecting failed nodes, and replaying the computation from sources or from persisted intermediate results [19]. Apache Spark, in contrast, improves the recovery by replaying from the sources through what is called lineage recovery [66]. A similar idea is used in a dynamic dataflow system within Ray [64]. This paper focused on the ABS protocol used in Apache Flink, which, in contrast to previous works, uses an asynchronous checkpointing technique [12]. It has been proven to provide high performance and has since been widely adopted [58]. The current version of Apache Flink’s runtime offers an opt-in feature for “unaligned checkpoints”, which allow the checkpoint markers to be treated at a higher priority, decreasing the end-to-end latency at the cost of some overhead as buffered events may become part of the snapshots [23]. Other adaptations of the Flink protocol include Clanos [59], which logs the nondeterminism to facilitate faster partial recovery after failures. Failure recovery remains an open research topic, as it has great impact on the performance characteristics of fault-tolerant systems [58].

8 Conclusions and Future Work

This paper studies failure transparency of stateful dataflow systems. We propose a novel definition of failure transparency for programming models expressed in small-step operational semantics. For the definition of failure transparency we introduce observational explainability, a notion which resembles refinement but on the level of observations of executions. We provide an implementation model of a stateful dataflow system using the Asynchronous Barrier Snapshotting protocol in a small-step operational semantics, and prove that the model is failure transparent and guarantees liveness.

In future work, we plan to implement a fully verified implementation of a stateful dataflow system based on the semantics presented in this paper, starting from our Coq mechanization. Furthermore, we would like to apply our definitions to existing related work.

References

- 1 Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991. doi:[10.1016/0304-3975\(91\)90224-P](https://doi.org/10.1016/0304-3975(91)90224-P).
- 2 Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, 2015. doi:[10.14778/2824032.2824076](https://doi.org/10.14778/2824032.2824076).
- 3 Bowen Alpern and Fred B. Schneider. Defining liveness. *Inf. Process. Lett.*, 21(4):181–185, 1985. doi:[10.1016/0020-0190\(85\)90056-0](https://doi.org/10.1016/0020-0190(85)90056-0).
- 4 Joe Armstrong. Erlang—a survey of the language and its industrial applications. In *Proc. INAP*, volume 96, pages 16–18, 1996.
- 5 Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent programming in ERLANG*. Prentice Hall, 1993.
- 6 Giorgio Audrito, Roberto Casadei, Ferruccio Damiani, Guido Salvaneschi, and Mirko Viroli. The exchange calculus (XC): A functional programming language design for distributed collective systems. *J. Syst. Softw.*, 210:111976, 2024. doi:[10.1016/J.JSS.2024.111976](https://doi.org/10.1016/J.JSS.2024.111976).
- 7 Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. In Fatma Özcan, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 13–24. ACM, 2005. doi:[10.1145/1066157.1066160](https://doi.org/10.1145/1066157.1066160).

- 8 Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S. Meiklejohn. Durable functions: semantics for stateful serverless. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–27, 2021. doi:10.1145/3485510.
- 9 Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming* (2. ed.). Springer, 2011. doi:10.1007/978-3-642-15260-3.
- 10 Paris Carbone. *Scalable and Reliable Data Stream Processing*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2018. URL: <https://nbn-resolving.org/urn:nbn:se:kth-diva-233527>.
- 11 Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State management in Apache Flink®: Consistent stateful distributed stream processing. *Proc. VLDB Endow.*, 10(12):1718–1729, 2017. URL: <http://www.vldb.org/pvldb/vol10/p1718-carbone.pdf>, doi:10.14778/3137765.3137777.
- 12 Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. Lightweight asynchronous snapshots for distributed dataflows. *CoRR*, abs/1506.08603, 2015. arXiv: 1506.08603.
- 13 Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015. URL: <http://sites.computer.org/debull/A15dec/p28.pdf>.
- 14 K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985. doi:10.1145/214451.214456.
- 15 Alvin Cheung, Natacha Crooks, Joseph M. Hellerstein, and Mae Milano. New directions in cloud programming. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021. URL: http://cidrdb.org/cidr2021/papers/cidr2021_paper16.pdf.
- 16 Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind Kami: a platform for high-level parametric hardware specification and its modular verification. *Proc. ACM Program. Lang.*, 1(ICFP):24:1–24:30, 2017. doi:10.1145/3110268.
- 17 David Cunningham, David Grove, Benjamin Herta, Arun Iyengar, Kiyokuni Kawachiya, Hiroki Murata, Vijay A. Saraswat, Mikio Takeuchi, and Olivier Tardieu. Resilient X10: efficient failure-aware programming. In José E. Moreira and James R. Larus, editors, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, pages 67–80. ACM, 2014. doi:10.1145/2555243.2555248.
- 18 Gabriela Jacques da Silva, Fang Zheng, Daniel Debrunner, Kun-Lung Wu, Victor Dogaru, Eric Johnson, Michael Spicer, and Ahmet Erdem Sarıyüce. Consistent regions: Guaranteed tuple processing in IBM streams. *Proc. VLDB Endow.*, 9(13):1341–1352, 2016. doi:10.14778/3007263.3007272.
- 19 Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In Eric A. Brewer and Peter Chen, editors, *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150. USENIX Association, 2004. URL: <http://www.usenix.org/events/osdi04/tech/dean.html>.
- 20 Joscha Drechsler, Ragnar Mogk, Guido Salvaneschi, and Mira Mezini. Thread-safe reactive programming. *Proc. ACM Program. Lang.*, 2(OOPSLA):107:1–107:30, 2018. doi:10.1145/3276477.
- 21 E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002. doi:10.1145/568522.568525.
- 22 John Field and Carlos A. Varela. Transactional memory: a programming model for maintaining globally consistent distributed state in unreliable environments. 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 195–208. ACM, 2005. doi:10.1145/1040305.1040322.

- 23 The Apache Software Foundation. Unaligned checkpoints flip-76. <https://issues.apache.org/jira/browse/FLINK-14551>, 2020. Accessed on 2024-03-28.
- 24 Marios Fragkoulis, Paris Carbone, Vasiliki Kalavri, and Asterios Katsifodimos. A survey on the evolution of stream processing systems. *VLDB J.*, 33(2):507–541, 2024. doi:10.1007/S00778-023-00819-8.
- 25 Yupeng Fu and Chinmay Soman. Real-time data infrastructure at Uber. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 2503–2516. ACM, 2021. doi:10.1145/3448016.3457552.
- 26 Felix C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv.*, 31(1):1–26, 1999. doi:10.1145/311531.311532.
- 27 Jonathan Goldstein, Ahmed S. Abdelhamid, Mike Barnett, Sebastian Burckhardt, Badrish Chandramouli, Darren Gehring, Niel Lebeck, Christopher Meiklejohn, Umar Farooq Minhas, Ryan Newton, Rahee Peshawaria, Tal Zaccai, and Irene Zhang. A.M.B.R.O.S.I.A: providing performant virtual resiliency for distributed applications. *Proc. VLDB Endow.*, 13(5):588–601, 2020. doi:10.14778/3377369.3377370.
- 28 Philipp Haller, Heather Miller, and Normen Müller. A programming model and foundation for lineage-based distributed computation. *J. Funct. Program.*, 28:e7, 2018. doi:10.1017/S0956796818000035.
- 29 Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Sloboďová, Christopher Taylor, Vladimir A. Frolov, Erik Reeber, and Armaghan Naik. Replacing testing with formal verification in Intel CoreTM i7 processor execution engine validation. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 414–429. Springer, 2009. doi:10.1007/978-3-642-02658-4_32.
- 30 Konstantinos Kallas, Haoran Zhang, Rajeef Alur, Sebastian Angel, and Vincent Liu. Executing microservice applications on serverless, correctly. *Proc. ACM Program. Lang.*, 7(POPL):367–395, 2023. doi:10.1145/3571206.
- 31 Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David A. Cock, Philip Derrin, Dhammadika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 207–220. ACM, 2009. doi:10.1145/1629575.1629596.
- 32 Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7. Athens, Greece, 2011.
- 33 Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. 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 179–192. ACM, 2014. doi:10.1145/2535838.2535841.
- 34 Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977. doi:10.1109/TSE.1977.229904.
- 35 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978. doi:10.1145/359545.359563.
- 36 Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994. doi:10.1145/177492.177726.
- 37 Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998. doi:10.1145/279227.279229.

- 38 Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. URL: <http://research.microsoft.com/users/lamport/tla/book.html>.
- 39 Leslie Lamport and Stephan Merz. Auxiliary variables in TLA+. *CoRR*, abs/1703.05121, 2017. [arXiv:1703.05121](https://arxiv.org/abs/1703.05121).
- 40 Peter Alan Lee and Thomas Anderson. *Fault Tolerance*, pages 51–77. Springer Vienna, Vienna, 1990. [doi:10.1007/978-3-7091-8990-0_3](https://doi.org/10.1007/978-3-7091-8990-0_3).
- 41 Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009. [doi:10.1145/1538788.1538814](https://doi.org/10.1145/1538788.1538814).
- 42 Barbara Liskov. Distributed programming in Argus. *Commun. ACM*, 31(3):300–312, 1988. [doi:10.1145/42392.42399](https://doi.org/10.1145/42392.42399).
- 43 David E. Lowell. *Theory and practice of failure transparency*. PhD thesis, University of Michigan, USA, 1999. URL: <https://hdl.handle.net/2027.42/132190>.
- 44 David E. Lowell and Peter M. Chen. The theory and practice of failure transparency. Technical report, University of Michigan, 1999.
- 45 Nancy Lynch and Mark Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, 1989. Also available as MIT Technical Memo MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology.
- 46 Nancy A. Lynch and Eugene W. Stark. A proof of the Kahn principle for input/output automata. *Inf. Comput.*, 82(1):81–92, 1989. [doi:10.1016/0890-5401\(89\)90066-7](https://doi.org/10.1016/0890-5401(89)90066-7).
- 47 Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In Ahmed K. Elmagarmid and Divyakant Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 135–146. ACM, 2010. [doi:10.1145/1807167.1807184](https://doi.org/10.1145/1807167.1807184).
- 48 Yancan Mao, Zhanghao Chen, Yifan Zhang, Meng Wang, Yong Fang, Guanghui Zhang, Rui Shi, and Richard T. B. Ma. StreamOps: Cloud-native runtime management for streaming services in ByteDance. *Proc. VLDB Endow.*, 16(12):3501–3514, 2023. [doi:10.14778/3611540.3611543](https://doi.org/10.14778/3611540.3611543).
- 49 Monica Marcus and Amir Pnueli. Using ghost variables to prove refinement. In Martin Wirsing and Maurice Nivat, editors, *Algebraic Methodology and Software Technology, 5th International Conference, AMAST '96, Munich, Germany, July 1-5, 1996, Proceedings*, volume 1101 of *Lecture Notes in Computer Science*, pages 226–240. Springer, 1996. [doi:10.1007/BF0014319](https://doi.org/10.1007/BF0014319).
- 50 Ragnar Mogk, Joscha Drechsler, Guido Salvaneschi, and Mira Mezini. A fault-tolerant programming model for distributed interactive applications. *Proc. ACM Program. Lang.*, 3(OOPSLA):144:1–144:29, 2019. [doi:10.1145/3360570](https://doi.org/10.1145/3360570).
- 51 Suvarn Mukherjee, Nitin John Raj, Krishnan Govindraj, Pantazis Deligiannis, Chandramouleswaran Ravichandran, Akash Lal, Aseem Rastogi, and Raja Krishnaswamy. Reliable state machines: A framework for programming reliable cloud services. 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 18:1–18:29. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. [doi:10.4230/LIPIcs.ECOOP.2019.18](https://doi.org/10.4230/LIPIcs.ECOOP.2019.18).
- 52 Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 439–455. ACM, 2013. [doi:10.1145/2517349.2522738](https://doi.org/10.1145/2517349.2522738).
- 53 Daniel Patterson and Amal Ahmed. The next 700 compiler correctness theorems (functional pearl). *Proc. ACM Program. Lang.*, 3(ICFP):85:1–85:29, 2019. [doi:10.1145/3341689](https://doi.org/10.1145/3341689).
- 54 Kyriakos Psarakis, Wouter Zorgdrager, Marios Fragnoulis, Guido Salvaneschi, and Asterios Katsifodimos. Stateful entities: Object-oriented cloud applications as distributed dataflows. In Letizia Tanca, Qiong Luo, Giuseppe Polese, Loredana Caruccio, Xavier Oriol, and Donatella Firmani, editors, *Proceedings 27th International Conference on Extending Database Technology, EDBT 2024, Paestum, Italy, March 25 - March 28*, pages 15–21. OpenProceedings.org, 2024. [doi:10.48786/EDBT.2024.02](https://doi.org/10.48786/EDBT.2024.02).

- 55 Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. End-to-end verification of processors with ISA-Formal. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 42–58. Springer, 2016. doi:10.1007/978-3-319-41540-6_3.
- 56 Mehl A. Shah, Joseph M. Hellerstein, and Eric A. Brewer. Highly-available, fault-tolerant, parallel dataflows. In Gerhard Weikum, Arnd Christian König, and Stefan Deßloch, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 827–838. ACM, 2004. doi:10.1145/1007568.1007662.
- 57 Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop distributed file system. In Mohammed G. Khatib, Xubin He, and Michael Factor, editors, *IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010*, pages 1–10. IEEE Computer Society, 2010. doi:10.1109/MSST.2010.5496972.
- 58 George Siachamis, Kyriakos Psarakis, Marios Fragkoulis, Arie van Deursen, Paris Carbone, and Asterios Katsifodimos. CheckMate: Evaluating checkpointing protocols for streaming dataflows. *CoRR*, abs/2403.13629, 2024. doi:10.48550/arXiv.2403.13629.
- 59 Pedro F. Silvestre, Marios Fragkoulis, Diomidis Spinellis, and Asterios Katsifodimos. Clonos: Consistent causal recovery for highly-available streaming dataflows. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 1637–1650. ACM, 2021. doi:10.1145/3448016.3457320.
- 60 Jonas Spenger, Paris Carbone, and Philipp Haller. Portals: An extension of dataflow streaming for stateful serverless. In Christophe Scholliers and Jeremy Singer, editors, *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2022, Auckland, New Zealand, December 8-10, 2022*, pages 153–171. ACM, 2022. doi:10.1145/3563835.3567664.
- 61 Olivier Tardieu, David Grove, Gheorghe-Teodor Bercea, Paul Castro, Jaroslaw Cwiklik, and Edward A. Epstein. Reliable actors with retry orchestration. *Proc. ACM Program. Lang.*, 7(PLDI):1293–1316, 2023. doi:10.1145/3591273.
- 62 Aleksey Veresov, Jonas Spenger, Paris Carbone, and Philipp Haller. Failure transparency in stateful dataflow systems (technical report), 2024. arXiv:2407.06738.
- 63 John von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata studies*, 34(34):43–98, 1956.
- 64 Stephanie Wang, John Liagouris, Robert Nishihara, Philipp Moritz, Ujval Misra, Alexey Tumanov, and Ion Stoica. Lineage stash: fault tolerance off the critical path. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 338–352. ACM, 2019. doi:10.1145/3341301.3359653.
- 65 John H. Wensley. SIFT: software implemented fault tolerance. In *American Federation of Information Processing Societies: Proceedings of the AFIPS '72 Fall Joint Computer Conference, December 5-7, 1972, Anaheim, California, USA - Part I*, volume 41 of *AFIPS Conference Proceedings*, pages 243–253. AFIPS / ACM / Thomson Book Company, Washington D.C., 1972. doi:10.1145/1479992.1480025.
- 66 Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Steven D. Gribble and Dina Katabi, editors, *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28. USENIX Association, 2012. URL: <https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia>.

- 67 Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In Erich M. Nahum and Dongyan Xu, editors, *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*. USENIX Association, 2010. URL: <https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets>.
- 68 Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: fault-tolerant streaming computation at scale. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 423–438. ACM, 2013. doi: 10.1145/2517349.2522737.
- 69 Steffen Zeuch, Ankit Chaudhary, Bonaventura Del Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp M. Grulich, Sebastian Breß, Jonas Traub, and Volker Markl. The NebulaStream platform for data and application management in the internet of things. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020. URL: <http://cidrdb.org/cidr2020/papers/p7-zeuch-cidr20.pdf>.

Inductive Predicate Synthesis Modulo Programs

Scott Wesley 

Dalhousie University, Halifax, Canada

Maria Christakis 

TU Wien, Austria

Jorge A. Navas 

Certora, Seattle, WA, USA

Richard Trefler 

University of Waterloo, Canada

Valentin Wüstholtz

ConsenSys, Vienna, Austria

Arie Gurfinkel 

University of Waterloo, Canada

Abstract

A growing trend in program analysis is to encode verification conditions within the language of the input program. This simplifies the design of analysis tools by utilizing off-the-shelf verifiers, but makes communication with the underlying solver more challenging. Essentially, the analysis tools operates at the level of input programs, whereas the solver operates at the level of problem encodings. To bridge this gap, the verifier must pass along proof-rules from the analysis tool to the solver. For example, an analysis tool for concurrent programs built on an inductive program verifier might need to declare Owicky-Gries style proof-rules for the underlying solver. Each such proof-rule further specifies how a program should be verified, meaning that the problem of passing proof-rules is a form of invariant synthesis.

Similarly, many program analysis tasks reduce to the synthesis of pure, loop-free Boolean functions (i.e., *predicates*), relative to a program. From this observation, we propose Inductive Predicate Synthesis Modulo Programs (IPS-MP) which extends high-level languages with minimal synthesis features to guide analysis. In IPS-MP, unknown predicates appear under assume and assert statements, acting as specifications modulo the program semantics. Existing synthesis solvers are inefficient at IPS-MP as they target more general problems. In this paper, we show that IPS-MP admits an *efficient* solution in the Boolean case, despite being generally undecidable. Moreover, we show that IPS-MP reduces to the satisfiability of constrained Horn clauses, which is less general than existing synthesis problems, yet expressive enough to encode verification tasks. We provide reductions from challenging verification tasks – such as parameterized model checking – to IPS-MP. We realize these reductions with an efficient IPS-MP-solver based on SEAHORN, and describe a real-world application to smart-contract verification.

2012 ACM Subject Classification Software and its engineering → Software verification

Keywords and phrases Software Verification, Invariant Synthesis, Model-Checking

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.43

Related Version *Extended Version:* <https://arxiv.org/abs/2407.08455>

Supplementary Material *Software (Source Code):* <https://github.com/seahorn/seahorn>
archived at sw.h1:dir:f3193eafa8d1a172794d2230c2abe4da7275b1af

Funding *Maria Christakis:* supported by the Vienna Science and Technology Fund (WWTF) and the City of Vienna [Grant ID: 10.47379/ICT22007].

Richard Trefler: supported, in part, by a Discovery Grant (Individual) from the Natural Sciences and Engineering Research Council of Canada.



© Scott Wesley, Maria Christakis, Jorge A. Navas, Richard Trefler, Valentin Wüstholtz, and Arie Gurfinkel;

licensed under Creative Commons License CC-BY 4.0

38th European Conference on Object-Oriented Programming (ECOOP 2024).

Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 43; pp. 43:1–43:30

 Leibniz International Proceedings in Informatics

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

Arie Gurfinkel: supported, in part, by a Discovery Grant (Individual) from the Natural Sciences and Engineering Research Council of Canada.

1 Introduction

In recent years, many tools have emerged to verify C programs by leveraging the Clang/LLVM compiler infrastructure (e.g., [9, 60, 56, 54, 31]). These tools take as input C programs annotated with assumptions and assertions, and decide whether an assertion can be violated given that all assumptions are satisfied. One such tool is SEAHORN [31], which employs techniques from software model checking [42], abstract interpretation [32], and memory analysis [43] to enable efficient verification. Due to these features, many tool designers have started using annotated C code as an intermediate language to dispatch program analysis problems to SEAHORN (e.g., [55, 40, 4, 15, 18, 66]). In this setting, programs with specifications are transformed into C programs with assumptions and assertions, and then these C programs are analyzed using SEAHORN. The results obtained from SEAHORN are examined to draw conclusions about the input programs.

However, the flexibility afforded by C code as an intermediate language makes communication with the underlying verification algorithm more challenging. When SEAHORN is given a program to verify, it automatically applies various builtin proof-rules, such as induction for loops [3] and function summarization [42]. A tool designer has no control over how these rules are employed, nor is the developer able to introduce new proof-rules to SEAHORN. The goal of this paper is to extend SEAHORN with the language features required to communicate new declarative proof-rules to the underlying verification algorithm.

To illustrate this challenge, we consider SMARTACE [66], a tool that uses SEAHORN for modular Solidity smart-contract verification. In SMARTACE, each smart-contract is modeled by a non-terminating loop that executes a sequence of transactions¹. For SMARTACE to verify a smart-contract, it first requires an inductive invariant for the non-terminating loop, and a compositional invariant for each map² in the program. The discovery of an inductive invariant is automated by SEAHORN’s invariant inference capabilities. However, SEAHORN is unaware of the modular proof-rules used by SMARTACE, and therefore, the end-user must provide the compositional invariants manually. The authors of SMARTACE hypothesized [66] that if each proof-rule could be declared to SEAHORN, then SEAHORN could instruct the underlying verification algorithm to infer all invariants automatically. Inspired by this hypothesis, we first implemented compositional invariant synthesis in SEAHORN, and then discovered that our solution generalized to many program verification problems. Consequently, our solution forms a general-purpose framework well-suited to compositional invariant synthesis.

To illustrate this more general problem, consider a tool designer who wishes to use an off-the-shelf software verifier (e.g., SEAHORN) as the back-end to a new analysis framework (e.g., SMARTACE). Recall that many off-the-shelf verifiers rely on specialized solvers to discharge verification conditions, including solvers for Satisfiability Modulo Theories [12], Constrained Horn Clauses (CHCs) [37], or intermediate verification languages (e.g., [10, 26]). As depicted in Figure 1, an analysis framework built atop an off-the-shelf verifier takes as input a program with specifications, translates this program into the language of the verifier, and then uses the verifier to generate verification conditions for its specialized solver. Since

¹ Transactions in Solidity/Ethereum can be thought of as sequences of method invocations.

² In Solidity, maps are often used to store data for individual smart-contract users.

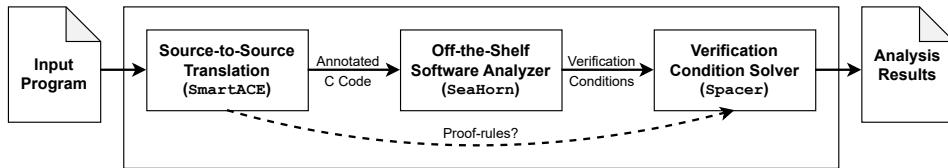


Figure 1 The architecture of an analysis framework built atop an off-the-shelf software verifier. Examples are given with respect to SMARTACE.

software verification is undecidable in general, it is often necessary for the tool designer to declare additional proof-rules for the solver. Example proof-rules include introducing predicate abstractions, suggesting modular abstractions for an array, and proposing modular decompositions for a parameterized system. However, it is challenging for the tool designer to communicate proof-rules to the solver – the former operates at the level of the input program, while the latter operates at the level of verification conditions. If a tool designer does attempt to encode proof-rules at the level of the input program, then these proof-rules are typically eliminated by optimizations from the verifier³, long before verification conditions are ever produced. That is, there is an impedance mismatch!

To bridge this gap, the verifier must pass proof-rules from the tool designer to the solver. Each proof-rule is associated with a set of invariants that the solver must find in order to prove the program correct. In other words, the invariants are declared by the proof-rules. Since these invariants span many classes (e.g., inductive, compositional, and object invariants), it is often the case that specialized invariant inference techniques cannot solve this problem. Instead, one must note that each proof-rule refines the invariants which the solver must synthesize. Consequently, one solution to the aforementioned impedance mismatch is to use synthesis techniques (e.g., [7, 25, 71, 59]). In particular, using synthesis allows the tool designer to declare proof-rules by specifying what invariants are to be synthesized at the level of the input program. This flexibility, however, comes at a price. General synthesis is significantly more expensive than verification [64]!

Our key contribution is a definition of *a new form of synthesis*, called *Inductive Predicate Synthesis Modulo Programs* (IPS-MP), that bridges the gap between flexible verification and efficient synthesis. Our *theoretical results* are two-fold, we show that: (a) IPS-MP reduces to satisfiability of CHCs, hence establishing that IPS-MP is a specialization of general synthesis [61, 63, 5, 41]; (b) for the special case of Boolean programs, IPS-MP is decidable with the same complexity as verification. We conjecture that the latter extends to other decidable models of programs (e.g., timed automata). Our *practical result* is to reduce a wide range of common proof-rules to IPS-MP. We show how IPS-MP guides inference of inductive invariants, class invariants, array invariants, and even modular parameterized model checking. In other words, IPS-MP is well-suited to many areas of program analysis. As a real-world application, we show that IPS-MP enables the full automation of SMARTACE.

Similar to existing synthesis frameworks, IPS-MP extends a programming language with unknowns. The language itself is unrestricted (i.e., it has loops, procedures, memory, etc.). However, the unknowns may only appear within `assume` and `assert` statements, denoting constraints on the strongest and weakest possible solutions, respectively. A solution to an IPS-MP problem is a mapping from each unknown to a Boolean predicate such that the resulting program is correct (i.e., satisfies all of its assertions). A high-level overview of

³ For example, a pure function with annotations may be optimized away by the Clang compiler.

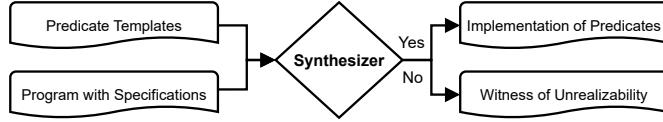


Figure 2 Overview of the IPS-MP problem.

IPS-MP is shown in Figure 2. Each problem instance consists of two components: (1) a program with its specification (described by assumptions and assertions), which contains calls to unknown predicates under assume and assert, and (2) the declarations of those predicates, which we refer to as *predicate templates*. Intuitively, a predicate template is a partial implementation with a number of unknown statements. A solution consists of a full implementation of each predicate, or a witness to unrealizability (i.e., a proof that a solution does not exist).

The reducibility of IPS-MP to CHC-solving motivates an efficient IPS-MP solver. We build on the SEAHORN framework (thus, our underlying language is the fragment of C supported by SEAHORN [31]), and integrate with two CHC solvers, namely SPACER [42], and ELDARICA [34]. Our empirical results on verification problems from various domains show that: (1) IPS-MP is effective at specifying verification strategies, (2) our implementation combined with existing CHC-solvers is highly efficient for linear arithmetic invariants, and (3) existing reductions to either general synthesis or specification inference are infeasible. Our evaluation focuses on general synthesis, rather than invariant inference, since the invariants in our benchmarks span many classes. To contextualize these results, we briefly review the state-of-the-art in synthesis.

State-of-the-art in synthesis. The general synthesis problem is the task of generating a program that satisfies a given specification. There are many general synthesis frameworks, e.g., Sketch [61], Rosette [63], SYGUS [5], and SEMGUS [41]. In Sketch and Rosette, users write programs with *holes*, representing unknowns. These holes are filled with predefined, loop-free expressions such that all program assertions are satisfied. SYGUS introduced a more language-agnostic approach to general synthesis. It generates loop-free *programs*, satisfying a given behavioral specification, from a potentially infinite language. Building on SYGUS, SEMGUS allows users to define pluggable semantics, thereby enabling synthesis of programs with loops. A distinguishing characteristic along this line of work is an emphasis on software development. In contrast, IPS-MP targets software verification and proof synthesis, which are theoretically simpler problems.

Specification synthesis (e.g., [21, 2, 53]) is another line of work that addresses a more specialized synthesis problem targeting program analysis, rather than software development. In specification synthesis, a program may call functions with unknown implementations. The goal is to synthesize specifications (e.g., the weakest specification for an unknown library procedure) that ensure the correctness of the calling program. Typically, a specification synthesizer imposes extra requirements, such as non-vacuity [53], maximality [2], or reachability [21], to ensure that solutions are reasonable. In contrast, the invariants synthesized by IPS-MP have constraints on both the strongest and weakest possible solutions, avoiding the need for additional (and often costly) requirements.

Of particular interest are the similarities and differences between IPS-MP and syntax-guided synthesis. In IPS-MP, program holes are filled by expressions from an unbounded language. To make this problem tractable, IPS-MP restricts Sketch and Rosette by requiring that holes only appear in partial predicates. Formally, this means that IPS-MP solving is

subsumed by non-linear constrained Horn clause solving. This restriction is crucial as it allows an IPS-MP solver to prove that a problem is unrealizable, unlike in Sketch or Rosette. Furthermore, IPS-MP differs from SYGUS and SEMGUS in that the behavioral specification is given with respect to a given program (in other words, modulo a given program), rather than through a separate logical specification. The program itself also places requirements on the holes, through assumptions and assertions, which is in contrast to specification synthesis.

In recent years, new extensions have been proposed to the Sketch framework. However, these extensions all generalize Sketch to more complex, and consequently less tractable, problems, whereas IPS-MP restricts Sketch to a more tractable problem which proves to be useful in the domain of program verification. To illustrate these gaps, we compare IPS-MP to PSKETCH [62], Synapse [16], Grisette [46], and MetaLift [13]. In the case of PSKETCH, both frameworks target the development of provably correct concurrent programs. However, PSKETCH focuses on inductive program verification in the presence of interleaving executions, whereas IPS-MP focuses on the verification of sequential code fragments via user-defined proof rules (e.g., the synthesis of compositional invariants in SMARTACE). In the case of Synapse, both tools aim to extend program synthesis problems with hints provided by an end-user. However, the nature of these hints is very different. In IPS-MP, the user introduces entirely new proof-rules, for which an underlying solver oversees the search for a solution. In contrast, the hints provided by an end-user to Synapse assign costs to solutions, for which the underlying solver tries to optimize. These hints do not allow the end-user to propose new proof-rules, and are suited to synthesis optimization rather than program verification. In the case of Grisette, a framework was proposed to programmatically generate and solve sketches. However, Grisette is based around bounded model-checking, whereas the IPS-MP problem targets unbounded model-checking and is, therefore, incomparable. More closely related is MetaLift, which makes use of the fact that inductive program verification can be reduced to syntax-guided synthesis. However, this verification program is not exposed to end-users. In particular, the assume and assert statements are hidden from end-users, and the end-user has no way to propose new placements for them. We conclude that IPS-MP is a novel synthesis problem.

Constrained Horn clauses. A prominent approach to verification is reduction to the satisfiability of CHCs, otherwise known as *verifier synthesis* [30]. While verifier synthesis does enable the flexible design of software verifiers, it does not address the issue of communicating proof-rules to the underlying solver. In invariant synthesis, the proof-rules are either chosen once and for all [30], or are implicit in the solving algorithm (e.g., [44, 65]). While we show that IPS-MP reduces to CHC-solving, our focus is on communicating new proof-rules to the solver via synthesis. Other solutions to IPS-MP might emerge in the future.

Contributions. This paper makes the following contributions:

1. Sec. 4 presents the novel IPS-MP problem which has many applications to both program analysis and software verification;
2. Sec. 5 shows that even though IPS-MP is undecidable in general, there exists an efficient solution modulo Boolean programs;
3. Sec. 6 provides reductions from important verification problems to IPS-MP;
4. Sec. 7 presents a solver for IPS-MP within SEAHORN. We demonstrate the effectiveness of our implementation compared to state-of-the-art synthesis frameworks CVC4 [11] (a SYGUS synthesizer) and HORN\$^{\text{SPEC}}\$ [53] (a specification synthesizer). We conclude that IPS-MP fills a gap not met by other synthesis frameworks.

All omitted proofs are found in the extended paper [68].

```

1 bool PRED_TEMPLATE Post(int x, int y) {
2   return synth(x, y); }
3 void main(int y) {
4   int x = 0;
5   assume(y > 0);
6   for (int i = 0; i < y; ++i) {
7     x+=1; }
8   assert(Post(x, y));
9   x = *; y = *; assume(Post(x, y));
10  assert(x == y); }
```

Figure 3 A simple example of the IPS-MP problem.

2 Overview

To illustrate the basics of IPS-MP, we start with an artificial example. For the moment, we focus on the language used in our presentation and the possible solutions to an IPS-MP problem. Realistic applications of IPS-MP, highlighting its importance, are presented later in this section.

Our example, shown in Figure 3, consists of a single function `main` that provides the context for a synthesis problem. The function `main` is written in a typical imperative language, with loops and function calls. We extend the language with two verification statements, `assume` and `assert`, with their usual semantics. In our example, `y` is initially positive, due to `assume(y > 0)` on line 5, and the program is correct if `assert(x == y)` on line 10 holds for all executions. That is, lines 5 and 10 provide a program specification. The goal of this example is to synthesize a pure expression `e` such that the program is correct after substituting `e` for each call to `Post`. To indicate that a predicate is a target for synthesis, the language is extended by the predicate template annotation `PRED_TEMPLATE` (line 1). Each predicate template is a pure, loop-free function whose body either returns `true`, or returns via a call to the special predicate `synth`. Each call to `synth` indicates a *hole* in the predicate implementation, and must be determined by a synthesizer. Each return of `true` places an *explicit constraint* on when the implementation must be true. In our example, `Post` always returns via a call to `synth` (line 2). In the rest of the program, a call to a partial predicate can only appear as an argument to either `assume` or `assert`. As described below, verification calls place *implicit constraints* on when the implementation must be true. Multiple calls to the same predicate are allowed. In our example, `Post` is called once under `assert` (line 8), and once under `assume` (line 9).

A *solution* to an IPS-MP problem is a mapping from each partial predicate p to a pure Boolean expression e over the arguments of p , such that if every call to `synth` in p is replaced by e , then the main program satisfies all of its assertions. If such a solution does not exist, the output is a *witness to unrealizability*, which is a mapping from each partial predicate p to a pure Boolean expression e over the arguments of p , which is both necessitated by the assertions placed on the partial predicate, and sufficient to violate an assertion that is part of the specification. In our example, there are many possible solutions. The weakest and strongest solutions are $post_{weak}(x, y) = (x = y)$ and $post_{strong}(x, y) = (y > 0 \wedge x = y)$. Each solution defines a corresponding predicate `Post` such that all assertions in the `main` program are satisfied. Intuitively, each call to `Post` under `assume` provides an implicit constraint on the weakest possible synthesized solution. Likewise, each call to `Post` under `assert` provides an implicit constraint on the strongest possible synthesized solution. Following this intuition, the example shows an application of IPS-MP to find an intermediate post-condition, over two variables `x` and `y`, that is true after the loop and is strong enough to ensure an assertion. This means that solving IPS-MP requires, in general, inferring inductive invariants for loops and summaries for functions.

To illustrate the case when synthesis is not possible, consider removing line 7 from Figure 3. Since `x` is not incremented, it will never equal `y`. However, `Post` cannot be mapped to `false`, since this violates the assertion on line 8. If `Post` is not `false`, then the assertion

on line 8 is reachable and will fail. Therefore, this IPS-MP problem is unrealizable. The witness to unrealizability is a mapping that sends `Post` to an expression over `x` and `y`, which is necessitated by the assertion on line 8 and violates the assertion on line 10. An example witness is $\text{synth}_{\text{witness}}(x, y) = (x = 0 \wedge y = 1)$.

This section continues with three important applications of IPS-MP. Sec. 2.1 presents a methodology to reduce verification problems to IPS-MP. For readers new to verification as synthesis, the standard example of inductive loop invariant inference can be found in the extended paper. Secs. 2.2, 2.3, and 2.4 extend on the techniques in the full paper to unify class invariant inference, array verification, and parameterized compositional model checking under a single synthesis framework. Sec. 2.5 discusses the benefits of predicate templates and explains why IPS-MP requires both assumptions and assertions of partial predicates. We note that the automation in SMARTACE is a special case of Sec. 2.3.

2.1 Methodology

In Figure 3, a single predicate (i.e., `Post`) represents a single unknown (i.e., the post-condition of a loop). This permits an IPS-MP solver to explore all relations between arguments (e.g. `x` and `y` of `Post`). When there are many variables, or large variable domains, the space of candidate solutions becomes very large. Restricting the syntactic structure of each unknown, referred to as its *shape*, helps to prune the search space. In general, an unknown can be split into cases (see Sec. 2.3), and the variables in each case can be partitioned (see the extended paper). Each partition is encoded by a unique predicate. Refining a predicate's shape prunes the candidate solution space, but may eliminate valid solutions.

Whenever an unknown is refined, the syntactic changes are reflected only where the unknown is assumed or asserted. The program remains unchanged otherwise. For this reason, in IPS-MP, it is convenient to separate unknowns from their shapes. In the context of program verification, this is accomplished with the following methodology. First, a proof-rule for the program of interest is reduced to assumptions and assertions on one or more unknowns. This is done once per proof-rule. Second, the shape of each unknown is refined using insight from the program. Third, the program is instrumented with assumptions and assertions. The instrumented program is an IPS-MP problem and is automatically solved by an IPS-MP solver. We illustrate this methodology using examples from object-oriented program analysis, array verification, and parameterized verification.

2.2 Class Invariant Inference as Synthesis

As a first example, we illustrate a reduction from class invariant inference to IPS-MP. In object-oriented programming, a class bundles together a data structure, its initialization procedure, and its operations. For example, the `Counter` class in Figure 4a accumulates values between 0 and some maximum value. The underlying data structure is a pair consisting of the current value, `pos`, and the maximum value, `max`. The initialization procedure on lines 3–6 first ensures that `_max` is positive, and then sets the current value to 0 and the maximum value to `_max`. The operations for `Counter` include `reset`, `capacity`, and `increment`. When `reset` is called, the current value is set back to 0. When `capacity` is called, the distance to the maximum value is returned. When `increment` is called, if `capacity` is greater than 0, then the current value is incremented and `true` is returned, else the current value is unchanged and `false` is returned.

The goal of this example is to prove that `drain` satisfies its assertions. The `drain` function takes an instance of `Counter` (in an arbitrary state), exhausts the counter's capacity, and then resets the counter to 0. The function is correct if `increment` always returns `true`

```

1 class Counter {
2     int max; int pos;
3     Counter(int _max) {
4         assume(_max > 0);
5         max = _max;
6         pos = 0;
7     }
8     void reset() { pos = 0; }
9     int capacity() {
10        return max - pos;
11    }
12    bool increment() {
13        if (pos >= max) return false;
14        pos += 1;
15        return true;
16    }
17    void drain(Counter o) {
18        while (o.capacity() > 0) {
19            assert(o.increment());
20            o.reset();
21            assert(o.capacity() > 0);
22        }
23    }
24 }
```

(a) The original program.

```

1 bool PRED_TEMPLATE CInv(int m, int p) {
2     return synth(m, p);
3 }
4 void main(void) {
5     if (*) {
6         Counter o(*);
7         assert(CInv(o.max, o.pos));
8     } else if (*) {
9         Counter o = *; assume(CInv(o.max, o.pos));
10        o.reset();
11        assert(CInv(o.max, o.pos));
12    } else if (*) {
13        Counter o = *; assume(CInv(o.max, o.pos));
14        o.increment();
15        assert(CInv(o.max, o.pos));
16    } else {
17        Counter o = *; assume(CInv(o.max, o.pos));
18        drain(o);
19    }
20 }
```

(b) The IPS-MP problem (using Figure 4a).

Figure 4 A program (see Figure 4a) which is correct relative to the choice of class invariant ($0 < \text{o}.max \wedge 0 \leq \text{o}.pos \leq \text{o}.max$), and a corresponding IPS-MP instance.

on line 17, and `capacity` always returns a positive value on line 19. Verifying these claims is non-trivial, as the correctness of `drain` depends on the possible states of `Counter`. For example, proving the assertion on line 17 requires the invariant ($0 \leq \text{max} - \text{pos}$).

A common approach to the modular analysis of object-oriented programs is class invariant inference (e.g., [24, 36, 1, 45]). A class invariant is a predicate that is true of a class instance after initialization, closed under the application of impure class methods, and sufficient to prove the correctness of the class [36]. In the case of `Counter`, the impure methods are `reset` and `increment`. Therefore, a class invariant for `Counter` must satisfy four requirements.

Figure 4b illustrates a technique to encode multiple cases in a single IPS-MP program. Intuitively, this program uses non-determinism to execute one of four possible cases. A case is selected on line 4 by a sequence of `if-else` statements, each with a non-deterministic condition. Even though the execution of each case is mutually exclusive, the IPS-MP solution must work in all cases. The cases in Figure 4b correspond to the requirements of a class invariant for `Counter`. To ensure that the class invariant holds after initialization, the first case initializes an instance of `Counter` with non-deterministic arguments, and then asserts that the instance satisfies the class invariant (lines 4–6). To ensure that the class invariant is closed with respect to `reset`, the second case selects an arbitrary instance of `Counter` (through non-determinism), assumes that this instance satisfies the class invariant, executes `reset`, and then asserts that the instance continues to satisfy the class invariant (lines 7–10). Similarly, the third case ensures that the class invariant is closed with respect to `increment` (lines 11–14). Finally, to ensure that the class invariant entails the correctness of `drain`, the fourth case selects an arbitrary instance of `Counter`, assumes that this instance satisfies the class invariant, and then calls `drain` with the instance as an argument (lines 15–17). This gives a program with unknowns, as required by the verification methodology.

Next, the shape of the class invariant is considered. In this example, we lack program-specific knowledge to help split the invariant into sub-cases. Furthermore, it would be futile to partition the invariant's arguments, as the invariant must relate `max` to `pos` (e.g., line 17 of Figure 4a requires that $0 \leq \text{max} - \text{pos}$). Therefore, `CInv(m, p)` is used as the shape of the invariant. In Figure 4b, `CInv` corresponds to the partial predicate on line 1. One solution to Figure 4b is the expression $(m > 0) \wedge (p \leq m)$ for the hole in `CInv`. To prove the correctness of `drain`, a synthesizer may also infer the invariant ($0 \leq \text{o}.pos \wedge \text{o}.pos \leq \text{o}.max$) for the loop on line 16 of Figure 4a.

```

1  bool PRED_TEMPLATE Inv3(int m, int v) {
2    if (m == 0 && v == 0) { return true; }
3    else { return synth(m, v); }
4  bool PRED_TEMPLATE Inv4(int m, int v){
5    if (m == 0 && v == 0) { return true; }
6    else { return synth(m, v); }
7  void main(int sid) {
8    int max = 0;
9    while (true) {
10      int id = *;
11      int x = *;
12      assume(id != x);
13      // int v = data[id];
14      int v = *; int u = *;
15      if (id == sid) { assume(Inv3(max,v)); }
16      else { assume(Inv4(max,v)); }
17      if (x == sid){ assume(Inv3(max,u)); }
18      else { assume(Inv4(max,u)); }
19      // Properties.
20      assert(v <= max);
21      if (id == sid) { assert(v == 0); }
22      // Update.
23      if (id != sid) { v += 1; }
24      if (v > max) { max = v; }
25      // data[id] = v;
26      if (id == sid) { assert(Inv3(max,v)); }
27      else { assert(Inv4(max,v)); }
28      if (x == sid) { assert(Inv3(max,u)); }
29      else { assert(Inv4(max,u)); }}}

```

(a) The original program.

(b) The IPS-MP problem.

Figure 5 A program (see Figure 5a) which is correct relative to the choice of array abstraction ($i = s \wedge v = 0 \vee (i \neq s \wedge 0 \leq v \leq \max)$), and a corresponding IPS-MP instance.

2.3 Verification of Array-Manipulating Programs as Synthesis

Consider the array-manipulating program in Figure 5a. This program initializes the array `data`, and then performs an unbounded sequence of updates to the cells of `data` while maintaining the maximum element of `data` in `max`. A special index, stored by `sid`, remains unchanged during execution. On lines 2–4, `data` is allocated and then zero-initialized. On line 5, `max` is set to 0, since the maximum element of a zero-initialized array is 0. On line 6, `sid` is set to an arbitrary index in `data`. The unbounded sequence of updates begins on line 7, when the program enters a non-terminating loop. During each iteration, an index is selected (lines 8–9), and if this index is not `sid`, then the corresponding cell in `data` is incremented by 1 (line 11). If the cell is incremented, then `max` is updated accordingly (line 12). Note that Figure 5a can be thought of as a simplified smart contract, where `data` is an *address mapping*, `sid` is an *address variable*, and each iteration of the loop is a *transaction*. For a more general presentation of smart contracts as array-manipulating programs, see [66].

The goal of this example is to prove two properties about the cells of `data`. The first property is that every cell of `data` is at most `max`. The second property is that `data[sid]` is always zero. It is not hard to see why these properties are true. For example, the first property is true since every cell of `data` is initially zero, and after increasing the value of a cell, `max` is updated accordingly. However, verifying these properties is challenging, since `data` has an arbitrary number of cells. One solution to this problem is to compute a summary for each cell of `data`, with respect to `max` and `sid`, and independent of `data`'s length. This summary is then used in place of each array access to obtain a new program with a finite number of cells. For simplicity, we assume that array accesses are in bounds, and that integers cannot overflow (i.e., are modeled as mathematical integers).

A common approach to obtain such a summary is to over-approximate the least fixed point of the program by an *abstract domain* that provides a tractable set of array cell partitions according to semantic properties (e.g., [29, 33, 20]). An alternative approach (followed here) is to compute a *compositional invariant* [49] for each cell of the array. A compositional (array) invariant is a predicate that is initially true of all cells in the array, and closed under every write to the array. Furthermore, a compositional invariant must be closed under *interference*, that is, if $i \neq j$ and the cell $\text{data}[i]$ is updated, then $\text{data}[j]$ continues to satisfy the compositional invariant. That is, a compositional invariant is assumed after each read and asserted after each write.

Using this approach, the program in Figure 5b is obtained. On line 10, an arbitrary index named `id` is selected, as in the original program. However, on lines 11–12, a second, *distinct* index named `x` is selected, to stand for a cell under interference. On lines 14–18, the compositional invariant is assumed, in place of reading the values at `data[id]` and `data[x]`. On lines 20–21, the two properties are asserted. If an arbitrary cell satisfies both properties, then every cell must satisfy both properties. On lines 23–24, the cell updates are performed as in the original program. On lines 26–29, the compositional invariant is asserted, in place of writing to `data[id]`. Note that lines 2–4 of Figure 5a do not appear in Figure 5b since the compositional invariant abstracts away the contents of `data`. This gives a program with unknowns, as required by the verification methodology.

Next, the shape of the compositional invariant is restricted. Observe that on line 11 of Figure 5a, the value written into `data` depends on whether the index is `sid`. This suggests that the compositional invariant has two cases that branch on whether `id` equals `sid`, namely $((\text{id} = \text{sid}) \wedge \text{Inv3}(\max, v)) \vee ((\text{id} \neq \text{sid}) \wedge \text{Inv4}(\max, v))$. In the IPS-MP encoding, both `Inv3` and `Inv4` correspond to partial predicates (see lines 1 and 4 in Figure 5b, respectively). The templates, on lines 2 and 5, correspond to the initialization rule for the invariant. Recall, however, that these templates are not strictly necessary. One alternative is to assert `Inv3(max, 0)` and `Inv4(max, 0)` before line 9, though this is not illustrated. Due to the branching shape of the invariant, each `assume` and `assert` statement must branch between the two partial predicates (see lines 14–18 and 26–29). Given Figure 5b, a synthesizer may find the expressions $(v == 0)$ for the hole in `Inv3`, and $(0 <= v) \wedge (v <= \max)$ for the hole in `Inv4`. By substitution, $((\text{id} = \text{sid}) \wedge (v = 0)) \vee ((\text{id} \neq \text{sid}) \wedge (0 \leq v) \wedge (v \leq \max))$. To verify `main`, a synthesizer may also infer the invariant $(0 \leq \max)$ for the loop at line 9.

2.4 Parameterized Verification as Synthesis

As a third example, we illustrate a reduction from parameterized verification to IPS-MP. This example considers two or more processes running in a ring network of arbitrary size. A ring network organizes processes into a single cycle, such that each process has a left and right neighbour [19]. In this ring, adjacent processes share a lock on a common resource. Processes are either *trying* to acquire a lock, or have acquired all locks and are in a *critical* section. Initially, all processes are trying and all locks are free. Each process runs the program in Figure 6a. The state of each process is given by `View` on line 3, and the transition relation of each process is given by `tr`⁴ on line 5. Since each process runs the same program with the same configuration of locks, the ring network is said to be *symmetric*.

⁴ For simplicity, `tr` is not deadlock-free as processes retain their locks until reaching their critical sections. However, the critical section can be reached any number of times without encountering a deadlock.

```

1 typedef enum { Free, Left, Right } Lock;
2 typedef enum { Try, Critical } State;
3 struct View {
4     Lock lhs; Lock rhs; State st; };
5 View tr(View v) {
6     bool held = v.lhs == Left &&
7             v.rhs == Right;
8     if (v.st == Critical) {
9         v.st = Try;
10        v.lhs = Free;
11        v.rhs = Free; }
12     else if (held) {
13         v.st = Critical; }
14     else if (v.lhs == Free) {
15         v.lhs = Left; }
16     else if (v.rhs == Free) {
17         v.rhs = Right; }
18     return v; }
```

(a) The process.

```

1 bool PRED_TEMPLATE RInv(
2     Lock l, State s, Lock r) {
3     if (l == Free && r == l && s == Try) {
4         return true;
5     } return synth(l, s, r); }
6 void main(struct View v) {
7     if (*) {
8         State otr = *;
9         assume(RInv(v.left, v.st, v.right));
10        assume(RInv(v.right, otr, v.left));
11        v = tr(v);
12        assert(RInv(v.left, v.st, v.right));
13        assert(RInv(v.right, otr, v.left));
14    } else {
15        assume(RInv(v.left, v.st, v.right));
16        bool held = v.left == Left &&
17                         v.right == Right;
18        assert(v.st != Critical || held); }}
```

(b) The IPS-MP problem (uses `tr`).

Figure 6 A process for a parameterized ring, and an IPS-MP problem that verifies the process. The process is correct relative to the compositional invariant $((v.lhs \neq \text{Left}) \vee (v.rhs \neq \text{Right})) \Rightarrow (v.st \neq \text{Critical})$, and the IPS-MP problem synthesizes the compositional invariant. Note that `Lock` and `State` are defined in Figure 6a using `typedef`, and that `otr` is a process under interference.

The goal of this example is to prove that if a process is in its critical section, then the process holds both adjacent locks. Following [49], this property is proven by computing an adequate compositional invariant for each process. An adequate compositional invariant is true of the initial state of each process, closed under the transition relation, closed under interference, and entails the property of interest. Remarkably, in a symmetric ring network, a compositional invariant can be computed by analyzing a ring with exactly two processes.

Using this approach, the program in Figure 6b is obtained. This program uses a non-deterministic `if` statement to split the analysis into two cases (line 7). The first case instantiates a two-process network using the compositional invariant (lines 8–10). Due to network symmetry, the left lock of the first process is the right lock of the second process, and vice versa. A single process in this network executes a transition (line 11), and then the closure of the compositional invariant is asserted for both processes (lines 12–13). The assertions on lines 12–13 ensure both closure under the transition relation and closure under interference, since only a single process transitioned. The second case instantiates a single process using the compositional invariant (line 15), and then asserts the property of interest (lines 16–18). Together, these cases define a compositional invariant. This gives a program with unknowns, as required by the verification methodology.

Next, the shape of the compositional invariant is considered. In this example, there is no motivation to split the invariant into cases. Furthermore, it would not make sense to partition the arguments of the invariant, since the state of a process is dependent on the combined state of its adjacent locks. Therefore, `RInv(l, s, r)` is assumed to be the shape of the invariant. In the IPS-MP encoding, `RInv` corresponds to the partial predicate on line 1. The template on line 3 ensures that the compositional invariant is true of the initial state of each process. As an alternative to a template, one can instead assert `RInv(Free, Try, Free)` before line 7. One solution to this problem is to fill the hole in `RInv` with the expression $(s == \text{Try}) \Rightarrow (l == \text{Left} \wedge r == \text{Right})$. Consequently, $((s \neq \text{Try}) \Rightarrow (l == \text{Left} \wedge r == \text{Right}))$.

```

1 bool PRED_TEMPLATE Inv(int x, int y) {
2   if (x + y == 5) { return true; }
3   else { return synth(x, y); }
4 void main(void) { ... }

```

```

1 bool PRED_TEMPLATE Inv(int x, int y) {
2   return synth(x, y); }
3 void main(void) {
4   int x = *; int y = *;
5   assume(x + y == 5);
6   assert(Inv(x, y));
7   ...

```

(a) Predicate template encoding.

(b) Assertion-based encoding.

Figure 7 The initial condition ($x + y = 5$) encoded using a predicate (see Figure 7a), and its equivalent encoding using an assertion (see Figure 7b).

2.5 Discussion

In Sec. 2.3, all explicit constraints were easily replaced by implicit constraints. However, explicit constraints can yield more succinct encodings. For example, consider the initial condition ($x + y = 5$). In Figure 7a, the condition is given as an explicit predicate template, and in Figure 7b, it is desugared as an assertion. To desugar the constraint, additional variables and assumptions are required.

In the examples presented so far, each IPS-MP problem places both assumptions and assertions on each partial predicate. All interesting IPS-MP problems follow this pattern. However, IPS-MP is well defined even if a partial predicate has only assumptions placed on it, only assertions placed on it, or neither. In these cases, the IPS-MP problem is trivial or reduces to a simpler problem.

If partial predicates only appear in assumptions, then the synthesized solution is never strengthened. In other words, the solution may be arbitrarily weak. This is an instance of specification synthesis. Usually, in specification synthesis, non-functional requirements are placed on each specification to ensure that a solution is “interesting” (e.g., [21, 2, 53]). Without these requirements, uninteresting solutions, such as `false`, are permitted. Since IPS-MP only places functional requirements on its solutions, this case is trivial and returning `false` from each predicate is always a solution (given a correct program).

If partial predicates only appear in assertions, then the synthesized solution is only ever strengthened. A solution in this case is an expression that subsumes all assertions placed on the predicate. However, an expression that evaluates to `true` subsumes all possible assertions. Therefore, this case is also trivial and returning `true` from each predicate is always a solution (given a correct program).

If partial predicates never appear in the program, then the synthesizer can select an arbitrary implementation for each predicate. However, if the synthesizer returns a solution, then the program must be correct relative to the solution. Therefore, if the program does violate an assertion, then the synthesizer must return a witness to unrealizability instead. Consequently, the output of the synthesizer indicates if the program is correct, and is equivalent to verification.

3 Background

This section recalls results from logic-based program verification. Sec. 3.1 reviews the key definitions of First Order Logic (FOL) and the Constrained Horn Clause (CHC) fragment of FOL. Sec. 3.2 introduces a programming language used throughout this paper. Sec. 3.3 recalls the connection between CHCs and program semantics through the weakest liberal precondition transformer.

3.1 First Order Logic and Constrained Horn Clauses

A *first order signature* Σ defines a set of predicates, a set of relations, and their respective arities. Given a set of variables \mathcal{V} , a *term* is either a variable from \mathcal{V} or an application of a relation in Σ to one or more terms. An *atom* is an application of a predicate in Σ to one or more terms. A *formula* joins atoms using standard logical connectives, existential quantification, and universal quantification. A formula is *quantifier-free* if it contains neither existential nor universal quantification. A formula is a *sentence* if all variable instances are quantified. Given a FO-formula φ , the formula $\varphi[x/y]$ is defined by substituting y for all free instances of x in φ . We write $\text{Term}(\Sigma, \mathcal{V})$ and $\text{QFFml}(\Sigma, \mathcal{V})$ for the sets of terms and quantifier-free formulas generated by Σ and \mathcal{V} .

A *FO-theory* \mathcal{T} is a deductively closed set of sentences over a signature Σ . A \mathcal{T} -*model* for a formula φ is an interpretation of each predicate, relation, and free variable in $\mathcal{T} \cup \{\varphi\}$ such that every formula in $\mathcal{T} \cup \{\varphi\}$ is true. If a \mathcal{T} -model exists for φ , then φ is *satisfiable*, otherwise, φ is *unsatisfiable*. In the case that all valid interpretations of \mathcal{T} are \mathcal{T} -models for φ , then φ is \mathcal{T} -valid and we write $\vdash_{\mathcal{T}} \varphi$. Furthermore, if each interpretation of a \mathcal{T} -model M can be expressed in some logical fragment \mathcal{F} , then M provides an \mathcal{F} -*solution* to φ .

Constrained Horn Clauses (CHCs) are a fragment of FOL determined by a FO-signature Σ and an set of predicates P . A CHC is a sentence of the form $\forall V \cdot \varphi \wedge p_1(\vec{x}_1) \wedge \dots \wedge p_k(\vec{x}_k) \Rightarrow h(\vec{y})$, where $\varphi \in \text{QFFml}(\Sigma, \mathcal{V})$ and $\{p_1, \dots, p_k, h\} \subseteq P$. For program semantics, it is useful to use v' to denote the value of a variable v after a program transition and $\text{keep}(W) := \bigwedge_{w \in W} w = w'$ to denote that variables $W \subseteq \mathcal{V}$ are unchanged during a transition. Given a set of variables $V = \{v_1, \dots, v_n\} \subseteq \mathcal{V}$, the set of variables $\{v'_1, \dots, v'_n\}$ is denoted V' . Likewise, given a formula φ over the variables in V , the formula $\varphi[v_1/v'_1] \dots [v_n/v'_n]$ over V' is denoted φ' .

3.2 Procedural Programming Language

Throughout this paper, we consider a simple procedural programming language, whose syntax is standard and can be found in the extended paper. We assume that all expressions are factored out by a FO-signature Σ , with variables from a set \mathcal{V} . That is, each expression is of the form $\text{QFFml}(\Sigma, \mathcal{V})$. The set of all programs in the language is denoted $\text{Progs}(\Sigma, \mathcal{V})$. For simplicity, types are omitted. In this language, a program has one or more procedures, with execution starting from `main`. Each procedure is written in an imperative language, including loops and procedure calls. The language is extended with a non-deterministic assignment (i.e., $*$), and verification statements `assume` and `assert`. The expressions in `assume` and `assert` can be either from $\text{QFFml}(\Sigma, \mathcal{V})$ or a call to a pure Boolean procedure, called a *predicate*. Predicates may only be called within `assume` or `assert` statements. Given a program $\mathcal{P} \in \text{Progs}(\Sigma, \mathcal{V})$, $\text{Procs}(\mathcal{P})$ denotes the procedures in \mathcal{P} . A special case is when all variables are Boolean.

► **Definition 1.** Let Σ_{Bool} denote a Boolean signature. A Boolean program is a tuple $(\text{Locs}, \text{GV}, \text{LV}, E)$ with $E = (\text{NE}, \text{CE}, \text{FE}, \text{AE}, \text{PE})$ and $V = \text{GV} \cup \text{LV}$ such that:

1. Locs is a finite set of control-flow locations with entry-point `main` $\in \text{Locs}$;
2. GV and LV are disjoint sets of local and global variables (respectively);
3. $\text{NE} \subseteq \text{Locs} \times \text{QFFml}(\Sigma_{\text{Bool}}, V \cup V') \times \text{Locs}$ is a set of normal edges, $\text{CE} \subseteq \text{Locs} \times \text{Locs} \times \text{Locs}$ is a set of call edges, $\text{FE} \subseteq \text{Locs} \times \text{Locs} \times \text{Locs}$ is a set of (partial predicate) call-under-`assume` edges, $\text{AE} \subseteq \text{Locs} \times \text{Locs} \times \text{Locs}$ is a set of (partial predicate) call-under-`assert` edges, and $\text{PE} \subseteq \text{Locs} \times \text{Locs}$ is a set of procedure summary edges;
4. If $(l_1, R, l_2) \in \text{NE}$, then l_2 is reachable from l_1 by updating the variables according to R and if $(l_{\text{call}}, l_{\text{in}}, l_{\text{ret}}) \in \text{CE}$, then l_{ret} is reachable from l_{call} by executing the procedure with entry location l_{in} ;

$$\begin{aligned} \text{ToCHC}(\mathcal{P}) &:= wlp(\mathcal{P}(\text{Main}), \top) \wedge \left(\bigwedge_{f \in \text{Procs}(\mathcal{P})} \text{ToCHC}(f) \right) \\ \text{ToCHC}(f(\vec{x}) \{ S; \text{return } \vec{e}; \}) &:= \forall \vec{x} \cdot (\vec{x} = \vec{x} \wedge f_{pre}(\vec{x}) \Rightarrow wlp(S, f_{sum}(\vec{x}, \vec{e}))) \\ \text{ToCHC}(p(\vec{x}) \{ \text{return } e; \}) &:= \forall \vec{x} \cdot p(\vec{x}) \Leftrightarrow e \end{aligned}$$

Figure 8 The partial correctness conditions for a program $\mathcal{P} \in \text{Progs}(\Sigma, \mathcal{V})$. This follows the presentation of [14].

5. If $(l_{\text{call}}, l_{\text{in}}, l_{\text{ret}}) \in FE$, then l_{ret} is reachable from l_{in} by assuming the partial predicate with entry location l_{call} and if $(l_{\text{call}}, l_{\text{in}}, l_{\text{ret}}) \in AE$, then l_{ret} is reachable from l_{in} by asserting the partial predicate with entry location l_{call} ;
6. If $(l_{\text{in}}, l_{\text{out}}) \in PE$, then the procedure with entry location l_{in} has exit location l_{out} .

A Boolean program consists of control-flow locations and edges between locations. Each procedure has a single entry location, l_{in} , and a single exit location, l_{out} , where $(l_{\text{in}}, l_{\text{out}}) \in PE$. The program enters at `main` $\in Locs$, and a special location $l_{\perp} \in Locs$ indicates failure. *Normal edges* connect control-flow locations within a procedure and represent non-procedural statements. For example, the statement `assert(e)` (where e is an expression) corresponds to two normal edges, $(l_1, e \wedge \text{keep}(GV \cup LV), l_2)$ and $(l_1, \neg e \wedge \text{keep}(GV \cup LV), l_{\perp})$. *Call edges* (optionally under assume or assert) connect locations in a caller's procedure and a callee's procedure by giving the call and return locations of the caller (l_{call} and l_{ret} , respectively), and the entry location for the callee (l_{in}). For simplicity, all procedures have the same local variables, and arguments are passed by global variables. The location l_{\perp} is assumed to have no outgoing edges.

The state of a Boolean program is a tuple (l, s) , where l is a location and s is an assignment to each Boolean variable. Initially, $l = \text{main}$ and s is an arbitrary assignment. For each normal edge (l_1, R, l_2) , a transition exists from (l_1, s_1) to (l_2, s_2) , if $s_1 \wedge R \wedge s'_2$ is valid. All call edges have the expected semantics.

3.3 Logical Program Verification

The Weakest Liberal Precondition (WLP) transformer gives logical semantics to imperative programs [23]. We write $wlp(S, Q)$ for the WLP of a statement S with respect to a post-condition Q . The WLP transformer for $\text{Progs}(\Sigma, \mathcal{V})$ is standard and can be found in the extended paper. Note that in this transformation, the $loop_{ln}$ predicate is an invariant for a loop at line ln .

The $wlp(-)$ transformer can be used to verify partial correctness for procedural programs. This is achieved through the $\text{ToCHC}(-)$ transformer in Figure 8. The $wlp(\mathcal{P}(\text{main}), \top)$ term asserts that `main` satisfies all assertions. For each procedure $f \in \text{Procs}(\mathcal{P})$, the term $\text{ToCHC}(f)$ asserts that f is correct for all inputs passed to f in every execution. Note that in Figure 8, f_{pre} collects inputs to f , and f_{sum} relates the inputs of f to the outputs of f . In the case that f is a predicate, f_{pre} and f_{sum} are omitted, since f is side-effect free. Together, $\text{ToCHC}(\mathcal{P})$ asserts that the program \mathcal{P} is correct for any execution starting from `main`. If $\text{ToCHC}(\mathcal{P})$ is satisfiable, then there exist loop invariants for \mathcal{P} such that \mathcal{P} satisfies all assertions [14]. Therefore, $\text{ToCHC}(\mathcal{P})$ can be used to verify \mathcal{P} . Furthermore, $\text{ToCHC}(\mathcal{P})$ is in the CHC fragment [14].

■ **Algorithm 1** Computes the least Boolean program summary (θ, σ) [8]. Follows the presentation of [17].

```

1 var  $(\theta, \sigma)$ ; // A program summary
2 var  $W$ ; // A map from  $Locs$  to a queued state
3 Func UpdateReach( $l, v$ ):
4    $s_{diff} \leftarrow v \wedge \neg \theta(l)$ ;
5   if  $s_{diff} \neq \perp$  then
6      $\theta(l) \leftarrow \theta(l) \vee s_{diff}$ ;
7      $W(l) \leftarrow W(l) \vee s_{diff}$ 
8 Func DoIntraproc( $V, NE, l_{wk}, s_{wk}$ ):
9   for  $(l_{wk}, R, l_2) \in NE$  do
10     $s_2 \leftarrow \text{elim}(s_{wk} \wedge R^*, V')$ ;
11     $s_2 \leftarrow s_2[V''/V']$ ;
12    UpdateReach( $l_2, s_2$ );
13 Func DoProcSum( $V, LV, PE, CE, l_{wk}, s_{wk}$ ):
14   for  $(l_{in}, l_{wk}) \in PE$  do
15     $s_{sum} \leftarrow \text{elim}(s_{wk}, LV \cup LV') \wedge \neg \sigma(l_{in})$ ;
16    if  $s_{sum} = \perp$  then continue;
17     $\sigma(l_{in}) \leftarrow \sigma(l_{in}) \vee s_{sum}$ ;
18    for  $(l_{call}, l_{in}, l_{ret}) \in CE$  do
19       $X \leftarrow s_{sum}^* \wedge \text{keep}(LV')$ ;
20       $s \leftarrow \text{elim}(\theta(l_{call}) \wedge X, V')[V''/V']$ ;
21      UpdateReach( $l_{ret}, s$ );
22 Func DoProcs( $V, LV, PE, CE, l_{wk}, s_{wk}$ ):
23   for  $(l_{wk}, l_{in}, l_{ret}) \in CE$  do
24     $s_{in} \leftarrow \text{elim}(s_{wk}, V \cup LV')[V'/V]$ ;
25    UpdateReach( $l_{in}, s_{in}$ );
26     $X \leftarrow \sigma(l_{in})^* \wedge \text{keep}(LV')$ ;
27     $s \leftarrow \text{elim}(s_{wk} \wedge X, V')[V''/V']$ ;
28    UpdateReach( $l_{ret}, s$ );
29 Func InitBoolReach( $Locs, PE$ ):
30   for  $l \in Locs$  do  $\theta(l) \leftarrow \perp$ ;
31   for  $(l_{in}, l_{out}) \in PE$  do  $\sigma(l_{in}) \leftarrow \perp$ ;
32    $\theta(\text{main}) \leftarrow \top; W(\text{main}) \leftarrow \top$ ;
33 Func ComputeBoolReach( $\mathcal{P}$ ):
34    $(Locs, GV, LV, (N, C, \emptyset, \emptyset, P)) \leftarrow \mathcal{P}$ ;
35    $V \leftarrow GV \cup LV$ ;
36   InitBoolReach( $Locs, P$ );
37   while  $\exists l_{wk} \in Locs \cdot W(l_{wk}) \neq \perp$  do
38     $s_{wk} \leftarrow W(l_{wk})$ ;  $W(l_{wk}) \leftarrow \perp$ ;
39    DoIntraproc( $V, N, l_{wk}, s_{wk}$ );
40    DoProcs( $V, LV, P, C, l_{wk}, s_{wk}$ );
41    DoProcSum( $V, LV, P, C, l_{wk}, s_{wk}$ );

```

Efficient procedures exist to prove that Boolean programs are correct. For example, *program summarization* simultaneously computes a summary θ from control-flow locations to input-to-reachable-state relations, and a summary σ from procedures to input-output relations. For a location $l \in Locs$, if $\theta(l) = \perp$, then l is unreachable. Therefore, a Boolean program \mathcal{P} is correct if and only if $\theta(l_\perp) = \perp$ in the least summary of \mathcal{P} [6]. Program summarization is defined in Def. 2⁵. The algorithm to compute θ is presented in full, for reuse in Sec. 5.1. For presentation, $\text{elim}(\varphi, W)$ denotes the existential elimination of W in φ .

► **Definition 2** ([6]). *A Boolean program summary for $(Locs, GV, LV, E)$, where $E = (NE, CE, \emptyset, \emptyset, PE)$ is a tuple (θ, σ) such that $Q = \text{QFFml}(\Sigma_{\text{Bool}}, V \cup V')$, $V = GV \cup LV$ and the following hold:*

1. $\sigma : Locs \rightarrow Q$ and $\theta : Locs \rightarrow Q$;
2. $\sigma(\text{main}) = \top$;
3. $\forall (l_1, R, l_2) \in NE \cdot \theta(l_1) \wedge R' \Rightarrow \theta(l_2)[V'/V'']$;
4. $\forall (l_{call}, l_{in}, l_{ret}) \in CE \cdot \theta(l_{call}) \wedge \sigma'(l_{in}) \wedge \text{keep}(LV') \Rightarrow \theta(l_{ret})[V'/V'']$;
5. $\forall (l_{call}, l_{in}, l_{ret}) \in CE \cdot \text{elim}(\theta(l_{call}), V \cup LV') \Rightarrow \theta(l_{in})$;
6. $\forall (l_{in}, l_{out}) \in PE \cdot \theta(l_{out}) \Rightarrow \sigma(l_{in})$.

`ComputeBoolReach` in Algorithm 1 is the standard algorithm to compute a least program summary. The algorithm works by iteratively applying the rules of Def. 2 until a fixed point is reached (we write $R^* := R[V'/V''][V/V']$). Termination is ensured by the finite-state of Boolean programs and the monotonicity of each rule. We extend on the algorithm `ComputeBoolReach` in Sec. 5.1.

⁵ To align with Algorithm 1, Def. 2 is non-standard but equivalent to [6].

```

1 bool Post(int x, int y) {
2   return x==y; }
3 void main(int y) {
4   assume(y>0); int x=0;
5   for (int i=0; i<y; ++i) { x+=1; }
6   assert(Post(x, y));
7   x=*>; y=*>; assume(Post(x, y)); }

```

(a) The program $\mathcal{P}[\Pi_{weak}]$.

```

1 bool Post(int x, int y) {
2   return (y>0) && (x==y); }
3 void main(int y) {
4   assume(y > 0); int x=0;
5   for (int i=0; i<y; ++i) { x+=1; }
6   assert(Post(x, y));
7   x=*>; y=*>; assume(Post(x, y)); }

```

(b) The program $\mathcal{P}[\Pi_{strong}]$.**Figure 9** Implementations of the simple program in Figure 3.

4 IPS-MP: Problem Definition

This section defines partial predicates and the IPS-MP problem. A *partial predicate* is a pure Boolean function without an implementation. A program \mathcal{P} is *open* if it contains a partial predicate p . An implementation for p is a Boolean expression e over the arguments of p . The program obtained by implementing p as `return e` is denoted $\mathcal{P}[p \leftarrow e]$. The set of all partial predicates in \mathcal{P} is written $\text{Partial}(\mathcal{P}) = \{p_1, \dots, p_k\}$. Given a function Π from $\text{Partial}(\mathcal{P})$ to pure Boolean expressions, we write $\mathcal{P}[\Pi]$ to denote $\mathcal{P}[p_1 \leftarrow \Pi(p_1)] \cdots [p_k \leftarrow \Pi(p_k)]$. The IPS-MP problem is to find a Π such that $\mathcal{P}[\Pi]$ is correct.

► **Example 3.** Recall program \mathcal{P} from Figure 3. Since `Post` is unimplemented in \mathcal{P} , then \mathcal{P} is an open program. Formally, $\text{Partial}(\mathcal{P}) = \{\text{Post}\}$. In Sec. 2, two implementations were proposed for `Post`, namely $(x = y)$ and $(y > 0 \wedge x = y)$. These implementations are represented by the mappings Π_{weak} and Π_{strong} from $\text{Partial}(\mathcal{P})$ to pure Boolean expressions such that $\Pi_{weak} : \text{Post} \mapsto (x = y)$ and $\Pi_{strong} : \text{Post} \mapsto (y > 0 \wedge x = y)$. The closed programs $\mathcal{P}[\Pi_{weak}]$ and $\mathcal{P}[\Pi_{strong}]$ are illustrated in Figure 9a and Figure 9b, respectively. ◀

► **Definition 4.** An Inductive Predicate Synthesis Modulo Programs (IPS-MP) problem is a tuple $(\mathcal{P}, \mathcal{T}, \Pi_0)$ such that $\mathcal{P} \in \text{Progs}(\Sigma, \mathcal{V})$ with first-order signature Σ and variable set \mathcal{V} , \mathcal{T} is a first-order theory, and $\Pi_0 : \text{Partial}(\mathcal{P}) \rightarrow \text{QFFml}(\Sigma, \mathcal{V})$ are predicate templates. A solution to $(\mathcal{P}, \mathcal{T}, \Pi_0)$ is a function $\Pi : \text{Partial}(\mathcal{P}) \rightarrow \text{QFFml}(\Sigma, \mathcal{V})$ such that $\mathcal{P}[\Pi]$ is correct relative to \mathcal{T} and $\forall p \in \text{Partial}(\mathcal{P}) \cdot \models_{\mathcal{T}} \Pi_0(p) \Rightarrow \Pi(p)$.

Assume that $(\mathcal{P}, \mathcal{T}, \Pi_0)$ is an IPS-MP problem with a solution Π . With respect to the IPS-MP overview in Figure 2, \mathcal{P} is a *program with specifications*, Π_0 is a collection of *predicate templates*, and Π is an *implementation of partial predicates*. The *witness to unrealizability* is discussed in Sec. 5. As an example of Def. 4, Figure 5b is restated as a formal IPS-MP problem.

► **Example 5.** This example restates Figure 5b as an IPS-MP problem $(\mathcal{P}, \Pi_0, \mathcal{T})$. The program \mathcal{P} is given by lines 7–29 of Figure 5b. Then $\text{Partial}(\mathcal{P}) = \{\text{Inv3}, \text{Inv4}\}$, since `Inv3` and `Inv4` are called on lines 15–16, but lack full implementations. From lines 1–6, $\Pi_0(\text{Inv3}) = \Pi_0(\text{Inv4}) = (m = 0 \wedge v = 0)$. Now, recall from Sec. 2.3 that all variables in Figure 5b are arithmetic integers. Therefore, \mathcal{T} is the theory of integer linear arithmetic. A solution to $(\mathcal{P}, \Pi_0, \mathcal{T})$ is Π such that $\Pi(\text{Inv3}) = (v = 0)$ and $\Pi(\text{Inv4}) = (0 \leq v \wedge v \leq m)$. ◀

5 Decidability of IPS-MP

This section considers the decidability of IPS-MP. Sec. 5.1 shows that IPS-MP is efficiently decidable in the Boolean case. Sec. 5.2 shows that IPS-MP is undecidable in general, but admits sound proof-rules for realizability and unrealizability.

5.1 The Case of Boolean Programs

This section shows that for Boolean programs, IPS-MP is decidable with the same time complexity as problem verification (i.e., polynomial in the number of program states). In contrast, general synthesis is known to have exponential time complexity in the Boolean case [64]. Therefore, IPS-MP modulo Boolean programs does in fact offer the benefits of general synthesis without the associated costs. To prove this result, we first extend Boolean program summaries (Def. 2) to programs with partial predicates. These new summaries are then used to extract solutions to IPS-MP (or witnesses to unrealizability). `Analyze` of Algorithm 2 extends on Algorithm 1 to compute these new summaries. The total correctness and time complexity of `Analyze` are proven in Cor. 9 and Thm. 7, respectively.

To simplify our presentation, we assume that all predicates are partial. In a Boolean program, each partial predicate has an entry location, but no edges nor exit location. This means that a standard summary can be obtained for a Boolean program with partial predicates by discarding all calls to partial predicates. Such a summary characterizes reachability, under the assumption that partial predicates are never called. From this summary, the arguments passed to each partial predicate under assert can be collected. For the program summary to be correct, the partial predicates must return true on these asserted arguments. If the partial predicate returns true on these asserted arguments, then for any call under assume using the same arguments, the program execution must continue to the next state. This procedure can then be repeated until a fixed point is obtained. This new *partial program summary* is defined formally in Def. 6.

► **Definition 6.** Let $\mathcal{P} = (\text{Locs}, \text{GV}, \text{LV}, (\text{NE}, \text{CE}, \text{FE}, \text{AE}, \text{PE}))$ be a Boolean program. A partial program summary for \mathcal{P} is a tuple (θ, σ, Π) such that:

1. $\Pi : \text{Partial}(\mathcal{P}) \rightarrow \text{QFFml}(\Sigma_{\text{Bool}}, \text{GV})$;
2. (θ, σ) is a program summary for $(\text{Locs}, \text{GV}, \text{LV}, (\text{NE}, \text{CE}, \emptyset, \emptyset, \text{PE}))$;
3. $\forall (l_{\text{call}}, l_{\text{in}}, l_{\text{ret}}) \in \text{AE} \cdot \theta(l_{\text{call}}) \Rightarrow \Pi'(l_{\text{in}})$;
4. $\forall (l_{\text{call}}, l_{\text{in}}, l_{\text{ret}}) \in \text{AE} \cdot \theta(l_{\text{call}}) \Rightarrow \theta(l_{\text{ret}})$;
5. $\forall (l_{\text{call}}, l_{\text{in}}, l_{\text{ret}}) \in \text{FE} \cdot \theta(l_{\text{call}}) \wedge \Pi'(l_{\text{in}}) \Rightarrow \theta(l_{\text{ret}})$.

The rules of Def. 6 follow directly from the preceding discussion. Rule 2 ensures that (θ, σ) is a program summary for \mathcal{P} after discarding all calls to partial predicates. Rules 3 and 4 collect the arguments passed to partial predicates under assert. Rule 5 advances the program state from calls to partial predicates under assume, according to the collected arguments. These steps are made operational by `Analyze` of Algorithm 2. Note that `Analyze` does not call `ComputeBoolReach` directly, and instead applies all rules in a single loop.

The termination of `Analyze` follows analogously to `ComputeBoolReach`. First, note that `Analyze` terminates if all work items have been processed. Each iteration of the loop at line 22 processes at least one work item. A state is added to the work list only if it has not yet been visited. The number of states is finite, since Boolean programs are finite-state. Therefore, `Analyze` must terminate with time polynomial in the number of program states. This is in contrast to general synthesis, which requires time exponential in the number of program states [64].

► **Theorem 7.** Let $\mathcal{P} = (\text{Locs}, \text{GV}, \text{LV}, E)$ with $E = (\text{NE}, \text{CE}, \text{FE}, \text{AE}, \text{PE})$ be a Boolean program. Then for each input (\mathcal{P}, Π_0) , `Analyze` of Algorithm 2 terminates in $O(n^2 m \cdot |\text{Locs}|)$ symbolic Boolean operations where $n = 2^{|\text{GV} \cup \text{LV}|}$ is the number of variable assignments and $m = |\text{NE} \cup \text{CE} \cup \text{FE} \cup \text{AE}|$ is the number of edges.

Algorithm 2 An extension of Algorithm 1 to solve IPS-MP for Boolean programs.

```

1 var  $(\theta, \sigma, \Pi)$ ; // A partial program summary    16 Func Init( $Locs, PE, \Pi_0$ ):
2 var  $W$ ; // A map from  $Locs$  to queued states    17   InitBoolReach( $Locs, PE$ );
3 Func DoAssumes( $V, LV, PE, FE, l_{wk}, s_{wk}$ ):      18   for  $l \in \text{Partial}(\mathcal{P})$  do  $\Pi(l) \leftarrow \Pi_0(l)$ ;
4   for  $(l_{wk}, l_{in}, l_{ret}) \in FE$  do                19 Func Analyze( $\mathcal{P}, \Pi_0$ ):
5      $s_{in} \leftarrow \text{elim}(s_{wk}, V \cup LV')[V'/V]$ ;  20   ( $Locs, GV, LV, (N, C, F, A, P)$ )  $\leftarrow \mathcal{P}$ ;
6     UpdateReach( $l_{ret}, \Pi(l_{in}) \wedge s_{in}$ );        21    $V \leftarrow GV \cup LV$ ; Init( $Locs, P, \Pi_0$ );
7 Func DoAsserts( $V, LV, PE, AE, l_{wk}, s_{wk}$ ):       22   while  $\exists l_{wk} \in Locs \cdot W(l_{wk}) \neq \perp$  do
8   for  $(l_{wk}, l_{in}, l_{ret}) \in AE$  do            23      $s_{wk} \leftarrow W(l_{wk})$ ;  $W(l_{wk}) \leftarrow \perp$ ;
9     UpdateReach( $l_{ret}, \Pi(l_{in}) \wedge s_{wk}$ );        24     DoIntraproc( $V, N, l_{wk}, s_{wk}$ );
10    UpdateReach( $l_{in}, \text{elim}(s_{wk}, V \cup LV')[V'/V]$ ); 25     DoProcs( $V, LV, P, C, l_{wk}, s_{wk}$ );
11 Func DoFuncSum( $V, LV, FE, AE, l_{wk}, s_{wk}$ ):        26     DoAssumes( $V, LV, P, F, l_{wk}, s_{wk}$ );
12   if  $l_{wk} \in \text{Partial}(\mathcal{P})$  then           27     DoAsserts( $V, LV, P, A, l_{wk}, s_{wk}$ );
13      $\Pi(l_{wk}) \leftarrow \Pi(l_{wk}) \vee s_{wk}$ ;        28     DoProcSum( $V, LV, P, C, l_{wk}, s_{wk}$ );
14     for  $(l_{call}, l_{wk}, l_{ret}) \in FE \cup AE$  do  29     DoFuncSum( $V, LV, F, A, l_{wk}, s_{wk}$ );
15       UpdateReach( $l_{ret}, \theta(l_{call}) \wedge s_{wk}$ );      30 Func BoolSynth( $\mathcal{P}, \Pi_0$ ):
16   |                                         31   Analyze( $\mathcal{P}, \Pi_0$ );
17   |                                         32   if  $\theta(l_{\perp}) = \perp$  then return  $(\checkmark, \Pi)$ ;
18   |                                         33   else return  $(\times, \Pi)$ ;

```

The correctness of `BoolSynth` follows from the correctness of `ComputeBoolReach` in [17]. Thm. 8 proves that `Analyze` extends `ComputeBoolReach` to obtain a least partial program summary. Cor. 9 proves that an IPS-MP solution (or a witness to unrealizability) can be extracted from a least partial program summary. Since `Analyze` terminates, this is a decision procedure for the Boolean case of IPS-MP.

► **Theorem 8.** *Let $\mathcal{P} = (Locs, GV, LV, E)$ be a Boolean program and Π_0 be a collection of predicate templates for \mathcal{P} . `Analyze` of Algorithm 2 computes a least partial program summary, (θ, σ, Π) , for \mathcal{P} such that $\forall p \in \text{Partial}(\mathcal{P}) \cdot \Pi_0(p) \Rightarrow \Pi(p)$.*

► **Corollary 9.** `BoolSynth` of Algorithm 2 decides IPS-MP for Boolean programs.

5.2 The General Case

This section presents sound proof-rules for the realizability and unrealizability of IPS-MP problems. These rules are shown to be instances of CHC-solving. To justify the reduction from IPS-MP to this undecidable problem, the general case of IPS-MP is also shown to be undecidable. First, assume that $(\mathcal{P}, \mathcal{T}, \Pi_0)$ is an IPS-MP problem. Recall that $\mathcal{P} \in \text{Progs}(\mathcal{F}, \mathcal{V})$ where \mathcal{F} is the FO-fragment of pure program expressions. A logical encoding of $(\mathcal{P}, \mathcal{T}, \Pi_0)$ is given by:

$$\text{CHCSynth}(\mathcal{P}, \Pi_0) := \text{ToCHC}(\mathcal{P}) \wedge \left(\bigwedge_{p \in \text{Partial}(\mathcal{P})} \forall \vec{x} \cdot (\Pi_0(p) \Rightarrow p(\vec{x})) \right)$$

The term `ToCHC`(\mathcal{P}) encodes verification conditions for \mathcal{P} , in which each partial predicate is unspecified. Calls to a partial predicate p , under assume and assert, provide constraints on the strongest and weakest possible solutions to `CHCSynth`(\mathcal{P}, Π_0). The clause $\forall \vec{x} \cdot (\Pi_0(p) \Rightarrow p(\vec{x}))$ then ensures the strongest solution to p subsumes $\Pi_0(p)$. Then a solution σ to `CHCSynth`(\mathcal{P}, Π) contains an implementation $\sigma(p)$ for each partial predicate p , that subsumes $\Pi_0(p)$ and ensures the correctness of \mathcal{P} (Thm. 10). Furthermore, if σ is an \mathcal{F} -solution, then each $\sigma(p)$ can be implemented in the programming language. On the other hand, if `CHCSynth`(\mathcal{P}, Π_0) is unsatisfiable, then for every choice of implementation Π satisfying Π_0 , the closed program $\mathcal{P}[\Pi]$ is incorrect (Thm. 11). Together, these theorems give sound proof rules for the realizability and unrealizability of $(\mathcal{P}, \mathcal{T}, \Pi_0)$. In practice, \mathcal{F} is chosen to be the same fragment used by the CHC-solver.

► **Theorem 10.** Let Σ be a first-order signature, \mathcal{V} be a set of variable symbols, $\mathcal{F} = \text{QFFml}(\Sigma, \mathcal{V})$, $\mathcal{P} \in \text{Progs}(\Sigma, \mathcal{V})$, and $(\mathcal{P}, \mathcal{T}, \Pi_0)$ be an IPS-MP problem. If σ is an \mathcal{F} -solution to $\text{CHCSynth}(\mathcal{P}, \Pi_0)$ relative to \mathcal{T} , then $\Pi : \text{Partial}(\mathcal{P}) \rightarrow \mathcal{F}$ such that $\Pi : p \mapsto \sigma(p)$ is a solution to $(\mathcal{P}, \mathcal{T}, \Pi_0)$.

► **Theorem 11.** If $(\mathcal{P}, \mathcal{T}, \Pi_0)$ is an IPS-MP problem and $\text{CHCSynth}(\mathcal{P}, \Pi_0)$ is \mathcal{T} -unsatisfiable, then $(\mathcal{P}, \mathcal{T}, \Pi_0)$ is unrealizable.

$\text{CHCSynth}(\mathcal{P}, \Pi_0)$ strengthens $\text{ToCHC}(\mathcal{P})$ by adding additional CHCs. Since $\text{ToCHC}(\mathcal{P})$ is a conjunction of CHCs, then $\text{CHCSynth}(\mathcal{P}, \Pi_0)$ is also a conjunction of CHCs. Therefore, a CHC solver can check the satisfiability and unsatisfiability of $\text{CHCSynth}(\mathcal{P}, \Pi_0)$. As a result, a CHC solver can find a solution to $(\mathcal{P}, \mathcal{T}, \Pi_0)$ (Thm. 10), or prove that the problem is unrealizable (Thm. 11).

► **Theorem 12.** $\text{CHCSynth}(\mathcal{P}, \Pi_0)$ is a CHC conjunction.

► **Example 13.** This example uses Thm. 10 to solve the IPS-MP problem in Figure 4b. The program in Figure 4b corresponds to the IPS-MP problem $(\mathcal{P}, \mathcal{T}, \Pi_0)$ where \mathcal{P} is the source code, \mathcal{T} is the theory of integer linear arithmetic, and $\Pi_0 : \text{CInv} \rightarrow \perp$. In this example we let \mathcal{F} be the fragment of linear inequalities of the variables $\{\mathbf{m}, \mathbf{p}\}$, where \mathbf{m} and \mathbf{p} are the arguments to CInv . Then our goal is to find an expression $e \in \mathcal{F}$ such that $\mathcal{P}[\text{CInv} \leftarrow e]$ is correct. According to Thm. 10, we can extract e from the output of a CHC-solver. The first step in this process is to construct the input $\text{CHCSynth}(\mathcal{P}, \Pi_0)$. To construct $\text{CHCSynth}(\mathcal{P}, \Pi_0)$ we must first construct the term $\text{ToCHC}(\mathcal{P})$. Recall that $\text{ToCHC}(\mathcal{P})$ encodes verification conditions for the program \mathcal{P} . Since \mathcal{P} is open (CInv is unimplemented), then CInv will be an unknown in $\text{ToCHC}(\mathcal{P})$. According to Sec. 3.3, $\text{ToCHC}(\mathcal{P})$ will consist of the verification conditions for $\mathcal{P}[\text{main}]$, along with a summary for each function in \mathcal{P} . We begin by constructing a summary for each method from the `Counter` object in \mathcal{P} . As described in Sec. 3.3, each predicate $f_{pre}(\mathbf{x})$ collects the inputs \mathbf{x} to a function f , and each predicate $f_{sum}(\mathbf{x}, \mathbf{e})$ each argument \mathbf{x} to a return value \mathbf{e} . For simplicity, we encode object state by passing member fields as arguments and return values. Redundant declarations are omitted.

$$\begin{aligned}\varphi_{Ctor} &:= \forall m \cdot \text{Counter}_{pre}(m) \Rightarrow ((m > 0) \Rightarrow \text{Counter}_{sum}(m, m, 0)) \\ \varphi_{Reset} &:= \forall m \cdot \forall p \cdot \text{reset}_{pre}(m, p) \Rightarrow \text{reset}_{post}(m, p, m, 0) \\ \varphi_{Cap} &:= \forall m \cdot \forall p \cdot \text{capacity}_{pre}(m, p) \Rightarrow \text{capacity}_{sum}(m, p, m - p \neq 0) \\ \varphi_{Incr} &:= \forall m \cdot \forall p \cdot \text{increment}_{pre}(m, p) \Rightarrow (((p \geq m) \Rightarrow \text{increment}_{sum}(m, p, \perp)) \wedge \\ &\quad ((p < m) \Rightarrow \text{increment}_{sum}(m, p + 1, \top)))\end{aligned}$$

Next, we construct a summary for the function `drain`. Note that, unlike the methods of `Counter`, the function `drain` contains a loop. As described in Sec. 3.3, loops are encoded using loop invariants with the loop at line n associated with an invariant $loop_n$. In our example, the loop at Line 16 of Figure 4a is associated with a loop invariant $loop_{13}$. Then the summary of `drain` is as follows.

$$\begin{aligned}\varphi_{Exit} &:= \text{capacity}_{pre}(p', m') \cdot \forall x \cdot (\text{capacity}_{sum}(p', m', x) \Rightarrow (x \wedge \text{drain}_{sum}(p, m, p', m'))) \\ \varphi_{Loop} &:= loop_{13}(p, m, x) \wedge \\ &\quad (((loop_{13}(p, m, x) \wedge x > 0) \Rightarrow (\text{capacity}_{pre}(p, m) \wedge \forall x \cdot (\text{capacity}_{sum}(p, mx) \Rightarrow loop_{13}))) \wedge \\ &\quad (((loop_{13}(p, m, x) \wedge x \leq 0) \Rightarrow \text{reset}_{pre}(p, m) \wedge \forall p' \cdot \forall m' \cdot (\text{reset}_{sum}(p, m, p', m') \Rightarrow \varphi_{Exit})) \\ \varphi_{Dr} &:= \forall p \cdot \forall m \cdot \text{drain}_{pre}(p, m) \Rightarrow (\text{capacity}_{pre}(p, m) \wedge \forall x \cdot (\text{capacity}_{sum}(p, m, x) \Rightarrow \varphi_{Loop}))\end{aligned}$$

Finally, we construct the verification conditions for `main`. Since `main` is the entry-point to \mathcal{P} , then `main` must be safe for all possible inputs. This means that `main` does not require a summary. The conditions are as follows.

$$\begin{aligned}
\varphi_{Main} := & \forall b_1 \cdot \forall b_2 \cdot \forall b_3 \cdot \\
& (b_1 = 1) \Rightarrow (\forall m \cdot Counter_{pre}(m) \wedge \forall m' \cdot \forall p \cdot (Counter_{sum}(m, m', p) \Rightarrow CInv(m', p)) \wedge \\
& (b_1 \neq 1 \wedge b_2 = 1) \Rightarrow (\forall m \cdot \forall p \cdot CInv(m, p) \Rightarrow \\
& (reset_{pre}(m, p) \wedge \forall m' \cdot \forall p' (reset_{sum}(p, m, p', m') \Rightarrow CInv(p', m')))) \wedge \\
& (b_1 \neq 1 \wedge b_2 \neq 1 \wedge b_3 = 1) \Rightarrow (\forall m \cdot \forall p \cdot CInv(m, p) \Rightarrow \\
& (increment_{pre}(m, p) \wedge \forall m' \cdot \forall p' (increment_{sum}(p, m, p', m') \Rightarrow CInv(p', m')))) \wedge \\
& (b_1 \neq 1 \wedge b_2 \neq 1 \neq b_3 = 1) \Rightarrow (\forall m \cdot \forall p \cdot CInv(m, p) \Rightarrow \\
& (drain_{pre}(m, p) \wedge \forall m' \cdot \forall p' (drain_{sum}(p, m, p', m') \Rightarrow \top)))
\end{aligned}$$

As outlined in Sec. 3.3, $\text{ToCHC}(\mathcal{P}) = \varphi_{Main} \wedge \varphi_{Ctor} \wedge \varphi_{Reset} \wedge \varphi_{Cap} \wedge \varphi_{Incr} \wedge \varphi_{Dr}$. Next, $\text{ToCHC}(\mathcal{P})$ is strengthened by the predicate template $\Pi_0(\text{CInv})$ to obtain $\text{CHCSynth}(\mathcal{P}, \Pi_0) = \text{ToCHC}(\mathcal{P}) \wedge (\forall m \cdot \forall p \cdot \perp \Rightarrow \text{CInv}(m, p))$. Clearly the term $\perp \Rightarrow \text{CInv}(m, p)$ is trivially satisfied. This is because the predicate template $\Pi_0(\text{CInv})$ is also trivial. In general, this need not be the case. Nonetheless, the term $\text{ToCHC}(\mathcal{P})$ is non-trivial. If $\text{ToCHC}(\mathcal{P})$ is provided to a CHC-solver, then the CHC-solver will return a solution σ containing the following components: expressions $\sigma(Counter_{pre})$, $\sigma(Reset_{pre})$, $\sigma(Capacity_{pre})$, $\sigma(Increment_{pre})$, and $\sigma(Drain_{pre})$, which over-approximate the inputs passed to each function; expressions $\sigma(Counter_{sum})$, $\sigma(Reset_{sum})$, $\sigma(Capacity_{sum})$, $\sigma(Increment_{sum})$, and $\sigma(Drain_{sum})$, which over-approximate the return values of each function; an expression $\sigma(loop_{13})$ which over-approximates the reachable states of the loop in `drain`; an expression $\sigma(\text{CInv})$ which describes a safe implementation for `CInv`. In one solution, $\sigma(loop_{13}) = (p \leq m \wedge (x \neq 0 \Rightarrow 0 < p) \wedge (x = 0 \Rightarrow 0 = p))$. This states that the counter is always in a valid position, and in position zero if and only if the capacity returns to zero. In such a solution, it is also possible that $\sigma(\text{CInv}) = (m > 0 \wedge p \leq m)$. Clearly $\sigma(\text{CInv})$ is an \mathcal{F} -solution since $\sigma(\text{CInv})$ is a conjunction of linear inequalities. Then by Thm. 10, $\Pi : \text{CInv} \rightarrow (m > 0 \wedge p \leq m)$ is a solution to $(\mathcal{P}, \mathcal{T}, \Pi_0)$ with $\mathcal{P}[\Pi]$ both closed and safe. \blacktriangleleft

Like CHC-solving, the general IPS-MP problem is also undecidable. This is because program verification reduces to IPS-MP. Intuitively, if a closed program \mathcal{P} is given to an IPS-MP solver, then a solution to the IPS-MP problem implies that \mathcal{P} is correct, and a witness to unrealizability implies that \mathcal{P} is incorrect. In this way, the halting problem also reduces to IPS-MP.

We show that IPS-MP is undecidable for linear integer arithmetic by reducing the halting problem for 2-counter machines to IPS-MP. Recall that a 2-counter machine is a program with a program counter and two integer variables [48]. The program has a finite number of locations, each with one of four instructions: (1) `inc(x)` increases the variable x by 1 and increment the program counter; (2) `dec(x)` decreases the variable x by 1 and increment the program counter; (3) `jump(x, i)` goes to location i if x is 0, else increments the program counter; (4) `halt()` halts execution of the program. The halting problem for 2-counter machines is known to be undecidable [48].

► **Theorem 14.** *The IPS-MP problem is undecidable for linear integer arithmetic.*

6 From Verification to Synthesis

This section establishes reductions of Sec. 2. Class invariant inference is proven directly. Array abstraction and symmetric ring verification are subsumed by a reduction from parameterized compositional model checking to IPS-MP. Loop invariant synthesis is proven in the extended paper. We write Σ for a first-order signature, \mathcal{V} for a set of variables, and Π_\perp for a collection of predicate templates which maps each predicate to \perp .

```

1  bool PRED_TEMPLATE Inv(int x, int y) {
2    return synth(x, y); }
3  void main(int br, int a, int b) {
4    if (br == 0) {
5      Cls ob = Cls(a);
6      assert(Inv(ob));
7    } else if (br == 1) {
8      Cls ob = *; assume(Inv(ob));
9      ob.f(a);
10     assert(Inv(ob));
11   } else if (br==2) {
12     Cls ob = *; assume(Inv(ob));
13     ob.g(a, b);
14     assert(Inv(ob));
15   } else if (br==3) {
16     Cls ob = *; assume(Inv(ob));
17     func(ob, a); }

```

(a) The input program.

(b) The IPS-MP reduction.

Figure 10 A reduction from class invariant inference to IPS-MP.

6.1 Class Invariant Inference

A *safe class invariant* is a predicate that is true of a class instance after initialization, closed under the execution of each impure class method, and sufficient to prove the correctness of a function taking class instances as arguments [36]. *Class invariant inference* asks to find a safe class invariant given a program. The inference problem is *intensional* if solutions are in the same logical fragment as assertions in the programming language [50]. A definition of (intensional) class invariant inference is found in Def. 15. In this definition, $\text{ToCHC}(f)$ relates the class invariant φ to a summary of each method f in \mathcal{P} , and f_{pre} is used to enforce that f is summarized. For simplicity, a class has two fields and two impure methods, each taking at most two arguments (Figure 10a). A generalization to m methods is not difficult. A generalization to n arguments follows immediately.

► **Definition 15.** A class invariant inference problem is a tuple $(\mathcal{P}, \mathcal{T})$ such that $\mathcal{P} \in \text{Progs}(\Sigma, \mathcal{V})$ is an open program as in Figure 10a and \mathcal{T} is a theory. A solution to $(\mathcal{P}, \mathcal{T})$ is a $\varphi \in \text{QFFml}(\Sigma, \{x, y\})$ such that the following are \mathcal{T} -satisfiable:

$$\begin{aligned}\psi_{Init} &:= \forall V (\text{Cls}_{pre}(a) \wedge \text{Cls}_{sum}(a, x, y) \Rightarrow \varphi) & \psi_{Close1} &:= \forall V (\varphi \wedge f_{sum}(x, y, a, x', y') \Rightarrow \varphi') \\ \psi_{Close2} &:= \forall V (\varphi \wedge g_{sum}(x, y, a, b, x', y') \Rightarrow \varphi') & \psi_{Suffic} &:= \forall V (\varphi \Rightarrow \text{func}_{sum}(x, y, a)) \\ \psi_{Sum} &:= \text{ToCHC}(\mathcal{P}) \wedge \forall V (\varphi \Rightarrow f_{pre}(x, y, a) \wedge g_{pre}(x, y, a, b) \wedge \text{func}_{pre}(x, y, a))\end{aligned}$$

► **Theorem 16.** Let $(\mathcal{P}, \mathcal{T})$ be a class invariant inference problem and \mathcal{P}' be the program obtained by adding `main` in Figure 10b to \mathcal{P} . Then Π is a solution to $(\mathcal{P}', \Pi_\perp, \mathcal{T})$ if and only if $\Pi(\text{Inv})$ is a solution to $(\mathcal{P}, \mathcal{T})$.

6.2 Reducing PCMC to IPS-MP

Parameterized compositional model checking (PCMC) is a framework to verify structures with arbitrarily many components (e.g., an array with arbitrarily many cells, or a ring with arbitrarily many processes) by decomposing the structure into smaller structures of fixed sizes [49]. Intuitively, each of these smaller structures is a view of the larger structure from the perspective of a single component. A proof of the larger structure is obtained by verifying each of the smaller structures, and showing that their proofs compose with one another [49]. If the number of smaller structures is finite (i.e., most perspectives are similar), then PCMC is applicable [49]. For example, in Sec. 2.3 and Sec. 2.4, the array and ring were highly symmetric, and therefore, all perspectives were similar.

```

1 bool PRED_TEMPLATE Inv(
2   int l, int s, int, r) {
3   if (init(l, s, r)) { return true; }
4   else { return synth(l, s, r); }
5 void main(int br, struct View v) {
6   if (br == 0) {
7     int inf = *;
8     assume(Inv(v.left, v.st, v.right));
9     assume(Inv(v.right, inf, v.left));
10    v = tr(v);
11    assert(Inv(v.left, v.st, v.right));
12    assert(Inv(v.right, inf, v.left));
13  } else if (br == 1) {
14    assume(Inv(v.left, v.st, v.right));
15    assert(property(v)); }

```

(a) The input program.

(b) The IPS-MP reduction.

Figure 11 A reduction from compositional ring invariant synthesis to IPS-MP. The state of each process and resource are both assumed to be integer values.

Once the larger structure has been decomposed, the proof of compositionality follows by inferring *adequate compositional invariants* for groups of similar components [49]. The number of compositional invariants, and the properties they must satisfy, depend on the decomposition. However, each property is one of initialization, closure, or non-interference. An initialization property states that a compositional invariant is true for the initial state of a component. A closure property states that a compositional invariant is closed under all transitions of its components. A non-interference property states that for any component c , if c satisfies its compositional invariant and an adjacent component (*also satisfying its compositional invariant*) performs a transition, then c continues to satisfy its compositional invariant after the transition. In addition, all composition invariants must be adequate in that they imply the correctness of the larger structure. To make the rest of this section concrete, we restrict ourselves to compositional ring invariants⁶. As in Sec. 6.1, the inference problem is assumed to be intensional. A formal definition of (intensional) compositional invariant inference is given in Def. 17⁷. Note that in Def. 17 ToCHC relates the compositional invariant φ to the summary of tr , tr_{pre} enforces that tr is summarized, and $\varphi_{\text{Inf}} := \varphi[l/r][s/i][r/l]$ is the compositional invariant applied to a process (r, i, l) .

► **Definition 17.** A compositional ring invariant (CRI) inference problem is a tuple $(\mathcal{P}, \mathcal{T})$ such that $\mathcal{P} \in \text{Progs}(\Sigma, \mathcal{V})$ is an open program as in Figure 11a and \mathcal{T} is a theory. A solution to $(\mathcal{P}, \mathcal{T})$ is a $\varphi \in \text{QFFml}(\Sigma, \{l, s, r\})$ such that the following are \mathcal{T} -satisfiable given $\varphi_{\text{Inf}} := \varphi[l/r][s/i][r/l]$:

$$\begin{aligned}
\psi_{\text{Init}} &:= \forall V (\text{init}(l, s, r) \Rightarrow \varphi) & \psi_{\text{Close}} &:= \forall V (\varphi \wedge \varphi_{\text{Inf}} \wedge \text{tr}_{\text{sum}}(l, s, r, l', s', r') \Rightarrow \varphi') \\
\psi_{\text{Adeq}} &:= \forall V (\varphi \Rightarrow \text{property}(l, s, r)) & \psi_{\text{Inf}} &:= \forall V (\varphi \wedge \varphi_{\text{Inf}} \wedge \text{tr}_{\text{sum}}(l, s, r, l', s', r') \Rightarrow \varphi_{\text{Inf}}) \\
\psi_{\text{Sum}} &:= \text{ToCHC}(\mathcal{P}) \wedge \forall V \cdot (\varphi \wedge \varphi_{\text{Inf}} \Rightarrow \text{tr}_{\text{pre}}(l, s, r))
\end{aligned}$$

► **Theorem 18.** Let $(\mathcal{P}, \mathcal{T})$ be a CRI inference problem, \mathcal{P}' be the program obtained by adding `main` of Figure 11b to \mathcal{P} , and Π_0 be the predicate template from Figure 11b. Then Π is a solution to $(\mathcal{P}', \Pi_0, \mathcal{T})$ if and only if $\Pi(\text{Inv})$ is a solution to $(\mathcal{P}, \mathcal{T})$.

⁶ Sec. 2.3 is a degenerate case. In this ring, processes communicate through locks. In an array, cells do not “communicate”.

⁷ In PCMC, a witness to unrealizability does not entail the incorrectness of a structure. Instead, no proof of correctness exists relative to the chosen decomposition.

Table 1 Performance of various solvers on IPS-MP benchmarks.

Type	Safe	Benchmarks			IPS-MP (SPACER)			IPS-MP (ELDARICA)			HORN-SPEC			CVC4			
		Buggy	Preds	Size	Time	TO	✓	Time	TO	MEM	✓	Time	UN	✓	TO	N/A	✓
Loop	7	7	9	179 KB	4	0	14	45	0	0	14	4	12	2	7	7	0
Class	6	6	6	694 KB	2	0	12	1449	0	0	12	—	12	0	6	6	0
Array	4	6	6	535 KB	4	0	10	230	0	0	10	—	10	0	4	6	0
Ring	2	3	2	197 KB	1	0	5	52	0	0	5	—	5	0	2	3	0
Proc	3	3	3	418 KB	2	0	6	4	0	0	6	—	6	0	3	3	0
SC	70	4	181	974 MB	6878	4	70	6717	53	12	9	—	—	—	—	—	—
Total	92	29	207	975 MB	6891	4	117	8497	53	12	56	4	45	2	22	29	0

7 Implementation and Evaluation

We have implemented an IPS-MP solver within the SEAHORN verification framework. SEAHORN takes as input a C program, and returns a CHC-based verification problem in the SMT-LIB format according to Sec. 3.3 [31]. We extend SEAHORN to recognize predicate templates. For each predicate, SEAHORN adds clauses to the verification conditions according to Sec. 5.2. Proofs of unrealizability are generated with the implementation of [28] found in SEAHORN. That is, proofs of unrealizability are already supported by SEAHORN.

The goal of our evaluation is to confirm that:

1. IPS-MP is practical for the reduction described in Sec. 6;
2. CHC-based solvers are more efficient than general synthesis solvers for IPS-MP instances;
3. The overhead incurred when using IPS-MP is tolerable.

Towards (1) and (2), we have collected 92 IPS-MP problems with linear integer arithmetic as the background theory (see **Safe** in Tab. 1). Of these benchmarks, 7 reflect loop invariant inference (and interpolation [47]), 6 reflect class invariant synthesis, 4 reflect array (and memory) abstraction, 2 reflect ring PCMC, 3 reflect procedure summarization, and 70 reflect parameterized analysis of smart-contract (SC) programs (see [66, 67]). The first 20 benchmarks were collected from research papers in the area of software verification. The remaining benchmarks, involving the parameterized analysis of SCs, were obtained by extending SMARTACE with support for IPS-MP. Of these 70 SC benchmarks, 62 are taken from real-world examples used to manage monetary assets [52]. To address question (3), we compare the performance of SMARTACE with and without IPS-MP, relative to these real-world examples. Note that the extension to SMARTACE was a routine exercise, due to the original design of SMARTACE. In particular, SMARTACE encodes all compositional invariants as predicates returning `true`, to then be refined manually by an end-user [67]. These predicates appear in assume and assert statements, as described in Sec. 2.4. Our extended version of SMARTACE can replace these predicates with predicate templates, yielding valid IPS-MP problems.

A summary of all benchmarks can be found in Tab. 1. As reflected by their size (see **Size** in Tab. 1) and total number of unknown predicates across all realizable instances (see **Preds** in Tab. 1), SCs are included to evaluate IPS-MP on large programs. When possible, benchmarks are drawn from prior works in program analysis (i.e., [39, 45, 58, 52]). To reflect unrealizability in IPS-MP, 29 faults have been injected in these benchmarks (see **Buggy** in Tab. 1). Further information can be found about the realizable real-world SC's in Tab. 2. Each SC in this table is associated with one or more safety properties (see **Props** in Tab. 2), which in turn, corresponds to a realizable IPS-MP instance. As before, **Preds** and **Size** indicate the total predicate count and size for these instances. All benchmarks are available at <https://doi.org/10.5281/zenodo.5083785>.

 **Table 2** Overhead of integrating IPS-MP-solving with SMARTACE.

Name	Contracts				Performance (Time)		
	Props	Preds	Size	✓	VERX [52]	SMARTACE (Manual) [66]	SMARTACE (IPS-MP)
Alchemist	3	12	36 MB	3	29	7	208
Brickblock	6	12	122 MB	6	191	13	1214
Crowdsale	9	27	76 MB	9	261	223	238
ERC20	9	27	45 MB	9	158	12	103
Melon	16	32	149 MB	16	408	30	979
PolicyPal	4	16	123 MB	4	20 773	26	3118
VUToken	5	22	319 MB	1	715	19	17
Zebi	5	14	45 MB	5	77	8	487
Zilliqa	5	10	54 MB	5	94	8	501
Total	62	172	969 MB	58	22 706	346	5685

To evaluate IPS-MP, we find the number of benchmarks that are solved by either of two state-of-the-art CHC solvers: ELDARICA [34] and SPACER [42]. To compare CHC solvers to general synthesis tools, we provide our benchmarks to a state-of-the-art specification synthesizer, HORNSPEC [53], and a state-of-the-art SyGUS solver, CVC4⁸ [11]. Since CVC4 solves SyGUS instances, which do not support proofs on unrealizability, then we only evaluate CVC4 on realizable benchmarks (see N/A in Tab. 1). Due to the size of each SC benchmark, we only ran the tools that could solve `Loop` through to `Proc` on these benchmarks. The results for each tool are reported in Tab. 1, where `TO` is the number of timeouts (after 30 minutes), `MEM` is the number of failures due to memory limits, `UN` is the number of benchmarks for which a tool returned `unknown`, `✓` is the number of benchmarks solved, and `Time` is the total time (in seconds) to find all solutions in a given set. In Tab. 2, the total time for SPACER is further broken down by SC (see SMARTACE (IPS-MP) in Tab. 2). For comparison, the verification times for VERX (an automated SC verifier with user-guided predicate abstraction [52]) and the original version of SMARTACE (see SMARTACE (Manual) in Tab. 2) are also provided. All evaluations were run on an Intel® Core i7® CPU @ 1.8GHz 8-core machine with 16GB of RAM running Ubuntu 20.04.

From this evaluation, we answer questions (1) through to (3) in the positive.

- As illustrated by Tab. 1, many examples of class invariant inference and compositional invariant inference (i.e., CLASS, ARRAY, RING, and SC) taken from the literature could be encoded using IPS-MP. In the case of SC, the generation of IPS-MP instances could be fully-automated using a modified version of SMARTACE. We conclude that IPS-MP is practical for the reductions described in Sec. 6.
- As shown in Tab. 1, all small benchmarks were solved by ELDARICA and SPACER, with average times under a minute. Furthermore, all but four SC benchmarks were solved by SPACER within a 30-minute timeout, with an average time of 96 seconds. Upon closer inspection, we found that SPACER would fail to solve these four examples, and would return `unknown` after approximately one hour. However, CVC4 failed to solve any SC benchmarks within 30-minutes. Therefore, we conclude that CHC-based IPS-MP-solving is effective for the reductions of Sec. 6, and can outperform general synthesis solutions. We note that HORNSPEC returned `unknown` on all but two benchmarks⁹.

⁸ To support CVC4, we convert each *realizable* problem from SMT-LIB format to the SyGUS input language.

⁹ The authors of HORNSPEC confirm this result though the cause is unknown.

3. As shown in Tab. 2, the IPS-MP version of SMARTACE incurred an average time overhead of 18x as compared to the manual version of SMARTACE. This should come as no surprise, since the manual version of SMARTACE achieved a verification time of under 3 seconds for 44 of the 62 properties with the help of user-provided compositional invariants. In these cases, a solving time as low as 60 seconds would correspond to an overhead of at least 20x. To better contextualize this overhead, we compare the verification time of IPS-MP version of SMARTACE to the verification time of VERX. We first note the outlying case of POLICYPAL, in which the IPS-MP version of SMARTACE achieves a speedup of over 6x. For the remaining SC’s, the IPS-MP version of SMARTACE fell within 1.3x of VERX on average. Since VERX is a specialized tool with less automation than the IPS-MP version of SMARTACE, we conclude that the overhead incurred by IPS-MP is tolerable in this particular real-world application. We note that in [52], only the “average” times were reported for VERX. It is unclear whether this is the average time to verify all properties, or an average across all properties. The authors of VERX were contacted, but were unable to provide the original data. For this reason, we assume conservatively that all times reported by VERX are total.

One limitation of the evaluation is its emphasis on SC verification. However, compositional SC verification is representative of compositional verification, as illustrated in [66]. We do acknowledge that design patterns specific to SC development might bias the benchmark set. We hope for this benchmark set to be expanded in future work.

Note, however, that we do not plan to benchmark our IPS-MP solver against invariant synthesis tools. Recall that our implementation simply extends SEAHORN with support for the IPS-MP synthesis language. In cases where the IPS-MP instance reduces to invariant synthesis, our extension is bypassed, and verification reduces to executing SEAHORN. Therefore, a direct comparison is not possible, and the evaluation results would not be meaningful. Furthermore, SEAHORN is a state-of-the-art program verifier with prior success in SV-COMP. Thus, SEAHORN is already known to perform well on invariant synthesis tasks.

An important direction for future work is to understand why CVC4 times out on all benchmarks. We hypothesize that the lack of a grammar in IPS-MP proves challenging for CVC4’s enumerative search. We also note that many of our benchmarks produce non-linear CHC’s, whereas the invariant synthesis track for SYGUS reduces to solving linear CHC’s.

8 Related Work

General program synthesis. As explained in Sec. 1, general synthesis engines (e.g., Sketch [61], Rosette [63], SYGUS [5], and SEMGUS [41]) are fundamentally different from IPS-MP. Among these frameworks, only SEMGUS can both solve synthesis problems and prove unrealizability. Similar to IPS-MP, SEMGUS reduces the synthesis problem to satisfiability of CHCs. However, this is where the similarities end. SEMGUS reduces synthesis to unsatisfiability and extracts solutions from the refutation proofs. In contrast, IPS-MP reduces to satisfiability and solutions are extracted from model of the CHCs. SEMGUS solves a more general problem, which comes at a high price both from a theoretical and practical perspective. We show that IPS-MP modulo Boolean programs can be solved in polynomial time (in the number of states), while SEMGUS lacks this guarantee. Existing SEMGUS solvers (e.g., MESSY [41]) synthesize programs from sets of candidates described using regular tree grammars. As a result, their CHCs use constraints over Algebraic Data Types to represent the grammar terms, which are harder to solve than either Boolean or linear arithmetic constraints. Only Sketch and Rosette are “modulo programs”, but do not allow loops nor recursion.

Specification synthesis. Specification synthesis solves the problem of finding specifications for unknown procedures which enable the verification of a given program (e.g., [21, 2, 53]). Unlike IPS-MP, specification synthesis is under-specified. Trivial specifications such as *false* are often sufficient but undesirable. As a consequence, many tools aim to synthesize either *weakest* (i.e., maximal) or non-vacuous solutions. In IPS-MP, any solution is valid as long as it satisfies all program assertions. In Sec. 7, we also compare our IPS-MP solver with HORNSPEC [53] and demonstrate that HORNSPEC is unsuitable for IPS-MP.

Data-driven invariant generation. Multiple approaches have been proposed (e.g., [27, 71, 59, 69, 57, 35]) that rephrase loop invariant synthesis as a learning problem. Recent work has extended these techniques to parameterized verification [70]. Often, these techniques require problem-specific biases to learn useful invariants (e.g., [59, 69, 57, 70]). Furthermore, these techniques lack the complexity bounds of decidable verification. In contrast, IPS-MP is problem-agnostic, and achieves the same complexity as verification in the Boolean case. Adapting data-driven techniques to IPS-MP-solving is an interesting future direction.

Constrained Horn clauses. In recent years, CHC-solvers have become a common tool for verification and synthesis problems. Examples include SEAHORN [31], SEMGUUS [41], and HORNSPEC [53]. The connection between CHCs and verification has long been explored in the CLP community (e.g., [38, 51, 22]). This direction was popularized again by the work of Rybalchenko et al. [30]. According to the annual CHC-COMP competition¹⁰, SPACER [42] and ELDARICA [34] are the most effective general-purpose CHC-solvers.

9 Conclusion

We proposed IPS-MP, a novel synthesis problem suitable for solving a wide range of verification problems, such as invariant inference and verification of parameterized systems. To demonstrate the relevance of IPS-MP, we provided three reductions from classic verification problems to IPS-MP. To highlight IPS-MP’s practicality, we proposed a solution that effectively leverages off-the-shelf CHC solvers and implemented it in the SEAHORN verification framework. Our evaluation demonstrates the effectiveness of CHC solvers in solving IPS-MP when compared with general synthesis tools such as HORNSPEC and CVC4.

Finally, we demonstrated that the interesting instance of IPS-MP for Boolean programs is efficiently decidable, whereas the general instance is undecidable. Despite this, the general instance of IPS-MP is theoretically simpler than general synthesis, and thus, warrants specialized solvers. In future work, we plan to study other instances of IPS-MP, such as IPS modulo timed automata. We further suspect that IPS-MP will enable new practical applications of PCMC.

References

- 1 Aneesh Aggarwal and Keith H. Randall. Related field analysis. In *PLDI*, pages 214–220. ACM, 2001.
- 2 Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. Maximal specification synthesis. In *POPL*, pages 789–801. ACM, 2016.

¹⁰ <https://chc-comp.github.io>.

- 3 Aws Albarghouthi, Yi Li, Arie Gurfinkel, and Marsha Chechik. Ufo: A framework for abstraction- and interpolation-based software verification. In *CAV*, volume 7358 of *LNCS*, pages 672–678. Springer Berlin Heidelberg, 2012.
- 4 Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. SAFEVM: a safety verifier for Ethereum smart contracts. In *ISSTA*, pages 386–389. ACM, 2019.
- 5 Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In Maximilian Irlbeck, Doron A. Peled, and Alexander Pretschner, editors, *Dependable Software Systems Engineering*, volume 40 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 1–25. IOS Press, 2015.
- 6 Rajeev Alur, Ahmed Bouajjani, and Javier Esparza. Model checking procedural programs. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 541–572. Springer Cham, 2018.
- 7 Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. SyGuS-Comp 2016: Results and analysis. In *SYNT@CAV*, volume 229 of *EPTCS*, pages 178–202, 2016.
- 8 Thomas Ball and Sriram K. Rajamani. Bebop: A path-sensitive interprocedural dataflow engine. In *PASTE*, pages 97–103. ACM, 2001.
- 9 Jiri Barnat, Lubos Brim, Milan Češka, and Petr Ročkai. DiVinE: Parallel distributed model checker. In *PDMC-HIBI*, pages 4–7. IEEE, 2010.
- 10 Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCOD*, volume 4111 of *LNCS*, pages 364–387. Springer Berlin Heidelberg, 2005.
- 11 Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *CAV*, volume 6806 of *LNCS*, pages 171–177. Springer Berlin Heidelberg, 2011.
- 12 Clark W. Barrett and Cesare Tinelli. Satisfiability modulo theories. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 305–343. Springer Cham, 2018.
- 13 Sahil Bhatia, Sumer Kohli, Sanjit A. Seshia, and Alvin Cheung. Building code compilers for domain-specific languages using program synthesis. In *ECOOP*, volume 263 of *LIPICS*, pages 38:1–38:30. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.
- 14 Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation II*, volume 9300 of *LNCS*, pages 24–51. Springer Berlin Heidelberg, 2015.
- 15 Roderick Bloem, Swen Jacobs, and Yakir Vizel. Efficient information-flow verification under speculative execution. In *ATVA*, pages 499–514. Springer Cham, 2019.
- 16 James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. Optimizing synthesis with Metasketches. In *POPL*, pages 775–788. ACM, 2016.
- 17 Sagar Chaki and Arie Gurfinkel. BDD-based symbolic model checking. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 219–245. Springer Cham, 2018.
- 18 Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. EzPC: Programmable and efficient secure two-party computation for machine learning. In *EuroS&P*, pages 496–511. IEEE, 2019.
- 19 K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1989.
- 20 Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL*, pages 105–118. ACM, 2011.

- 21 Ankush Das, Shuvendu K. Lahiri, Akash Lal, and Yi Li. Angelic verification: Precise verification modulo unknowns. In *CAV*, volume 9206 of *LNCS*, pages 324–342. Springer Berlin Heidelberg, 2015.
- 22 Giorgio Delzanno and Andreas Podelski. Model checking in CLP. In *TACAS*, volume 1579 of *LNCS*, pages 223–239. Springer Berlin Heidelberg, 1999.
- 23 Edsger Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.
- 24 Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington, USA, 2002.
- 25 Grigory Fedyukovich, Samuel J. Kaufman, and Rastislav Bodík. Sampling invariants from frequency distributions. In *FMCAD*, pages 100–107. IEEE, 2017.
- 26 Jean-Christophe Filliâtre and Andrei Paskevich. Why3—Where programs meet provers. In *ESOP*, volume 7792 of *LNCS*, pages 125–128. Springer Berlin Heidelberg, 2013.
- 27 Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. ICE: A robust framework for learning invariants. In *CAV*, volume 8559 of *LNCS*, pages 69–87. Springer Berlin Heidelberg, 2014.
- 28 Jeffrey Gennari, Arie Gurfinkel, Temesghen Kahsai, Jorge A. Navas, and Edward J. Schwartz. Executable counterexamples in software model checking. In *VSTTE*, volume 11294 of *LNCS*, pages 17–37. Springer, 2018.
- 29 Denis Gopan, Thomas W. Reps, and Shmuel Sagiv. A framework for numeric analysis of array operations. In *POPL*, pages 338–350. ACM, 2005.
- 30 Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popescu, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416. ACM, 2012.
- 31 Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The SeaHorn verification framework. In *CAV*, volume 9206 of *LNCS*, pages 343–361. Springer Berlin Heidelberg, 2015.
- 32 Arie Gurfinkel and Jorge A. Navas. Abstract interpretation of LLVM with a region-based memory model. In Roderick Bloem, Rayna Dimitrova, Chuchu Fan, and Natasha Sharygina, editors, *Software Verification*, volume 13124 of *LNCS*, pages 122–144. Springer, 2022.
- 33 Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In *PLDI*, pages 339–348. ACM, 2008.
- 34 Hossein Hojjat and Philipp Rümmer. The ELDARICA Horn solver. In *FMCAD*, pages 1–7. IEEE, 2018.
- 35 Kangjing Huang, Xiaokang Qiu, Peiyuan Shen, and Yanjun Wang. Reconciling enumerative and deductive program synthesis. In *PLDI*, pages 1159–1174. ACM, 2020.
- 36 Kees Huizing and Ruurd Kuiper. Verification of object oriented programs using class invariants. In *FASE*, volume 1783 of *LNCS*, pages 208–221. Springer Berlin Heidelberg, 2000.
- 37 Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *POPL*, pages 111–119. ACM, 1987.
- 38 Joxan Jaffar, Andrew E. Santosa, and Razvan Voicu. A CLP proof method for timed automata. In *RTSS*, pages 175–186. IEEE Computer Society, 2004.
- 39 Temesghen Kahsai, Rody Kersten, Philipp Rümmer, and Martin Schäf. Quantified heap invariants for object-oriented programs. In *LPAR*, volume 46 of *EPiC Series in Computing*, pages 368–384. EasyChair, 2017.
- 40 Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. ZEUS: analyzing safety of smart contracts. In *NDSS*. The Internet Society, 2018.
- 41 Jinwoo Kim, Qinheping Hu, Loris D’Antoni, and Thomas W. Reps. Semantics-Guided Synthesis. *Proc. ACM Program. Lang.*, 5(POPL):1–32, 2021.
- 42 Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. SMT-based model checking for recursive programs. In *CAV*, volume 8559 of *LNCS*, pages 17–34. Springer Berlin Heidelberg, 2014.
- 43 Jakub Kuderski, Jorge A. Navas, and Arie Gurfinkel. Unification-based pointer analysis without oversharing. In *FMCAD*, pages 37–45. IEEE, 2019.

- 44 Jérôme Leroux, Philipp Rümmer, and Pavle Subotic. Guiding Craig interpolation with domain-specific abstractions. *Acta Informatica*, 53(4):387–424, 2016.
- 45 Francesco Logozzo. Automatic inference of class invariants. In *VMCAI*, volume 2937 of *LNCS*, pages 211–222. Springer Berlin Heidelberg, 2004.
- 46 Sirui Lu and Rastislav Bodík. Grisette: Symbolic compilation as a functional programming library. *Proc. ACM Program. Lang.*, 7(POPL), 2023.
- 47 Kenneth McMillan. Interpolation and SAT-based model checking. In *CAV*, volume 2725 of *LNCS*, pages 1–13. Springer Berlin Heidelberg, 2003.
- 48 Marvin Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, 1967.
- 49 Kedar S. Namjoshi and Richard J. Trefler. Parameterized compositional model checking. In *TACAS*, volume 9636 of *LNCS*, pages 589–606. Springer Berlin Heidelberg, 2016.
- 50 Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer*. Undergraduate Topics in Computer Science. Springer, 2007.
- 51 Julio C. Peralta, John P. Gallagher, and Hüseyin Saglam. Analysis of imperative programs through analysis of constraint logic programs. In *SAS*, volume 1503 of *LNCS*, pages 246–261. Springer Berlin Heidelberg, 1998.
- 52 Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. VerX: Safety verification of smart contracts. In *S&P*, pages 1661–1677. IEEE, 2020.
- 53 Sumanth Prabhu, Grigory Fedyukovich, Kumar Madhukar, and Deepak D’Souza. Specification synthesis with constrained Horn clauses. In *PLDI*, pages 1203–1217. ACM, 2021.
- 54 Zvonimir Rakamarić and Michael Emmi. SMACK: Decoupling source language details from verifier implementations. In *CAV*, volume 8559 of *LNCS*, pages 106–113. Springer Cham, 2014.
- 55 Dan Rasin, Orna Grumberg, and Sharon Shoham. Modular verification of concurrent programs via sequential model checking. In *ATVA*, pages 228–247. Springer Cham, 2018.
- 56 Heinz Riener and Görschwin Fey. FAuST: A framework for formal verification, automated debugging, and software test generation. In *SPIN*, volume 13872 of *LNCS*, pages 234–240. Springer Berlin Heidelberg, 2012.
- 57 Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. CLN2INV: learning loop invariants with continuous logic networks. In *ICLR*. OpenReview.net, 2020.
- 58 Malte Hermann Schwerhoff. *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution*. PhD thesis, ETH Zurich, Switzerland, 2016.
- 59 Xujie Si, Aaditya Naik, Hanjun Dai, Mayur Naik, and Le Song. Code2Inv: A deep learning framework for program verification. In *CAV*, volume 12225 of *LNCS*, pages 151–164. Springer Berlin Heidelberg, 2020.
- 60 Carsten Sinz, Stephan Falke, and Florian Merz. A precise memory model for low-level bounded model checking. In *SSV*, page 7, USA, 2010. USENIX Association.
- 61 Armando Solar-Lezama. Program sketching. *Int. J. Softw. Tools Technol. Transf.*, 15(5-6):475–495, 2013.
- 62 Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. Sketching concurrent data structures. In *PLDI*, pages 136–148. ACM, 2008.
- 63 Emin Torlak and Rastislav Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI*, pages 530–541. ACM, 2014.
- 64 Moshe Y. Vardi. From verification to synthesis. In *VSTTE*, volume 5295 of *LNCS*, page 2. Springer Berlin Heidelberg, 2008.
- 65 Hari Govind VK, Yuting Chen, Sharon Shoham, and Arie Gurfinkel. Global guidance for local generalization in model checking. In Shuvendu K. Lahiri and Chao Wang, editors, *CAV*, volume 12225 of *LNCS*, pages 101–125. Springer Berlin Heidelberg, 2020.
- 66 Scott Wesley, Maria Christakis, Jorge A. Navas, Richard Trefler, Valentin Wüstholtz, and Arie Gurfinkel. Compositional verification of smart contracts through communication abstraction. In *Static Analysis*, volume 12913 of *LNCS*, pages 429–452. Springer Berlin Heidelberg, 2021.

43:30 Inductive Predicate Synthesis Modulo Programs

- 67 Scott Wesley, Maria Christakis, Jorge A. Navas, Richard Trefler, Valentin Wüstholtz, and Arie Gurfinkel. Verifying Solidity smart contracts via communication abstraction in SmartACE. In *VMCAI*, pages 425–449. Springer Cham, 2022.
- 68 Scott Wesley, Maria Christakis, Jorge A. Navas, Richard Trefler, Valentin Wüstholtz, and Arie Gurfinkel. Inductive predicate synthesis modulo programs (extended), 2024.
- 69 Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. Learning nonlinear loop invariants with gated continuous logic networks. In *PLDI*, pages 106–120. ACM, 2020.
- 70 Jianan Yao, Runzhou Tao, Ronghui Gu, Jason Nieh, Suman Jana, and Gabriel Ryan. DistAI: Data-driven automated invariant learning for distributed protocols. In *OSDI*, pages 405–421. USENIX Association, 2021.
- 71 He Zhu, Stephen Magill, and Suresh Jagannathan. A data-driven CHC solver. In *PLDI*, pages 707–721. ACM, 2018.

Type Tailoring

Ashton Wiersdorf  

University of Utah, Salt Lake City, UT, USA

Stephen Chang  

University of Massachusetts Boston, MA, USA

Matthias Felleisen  

Northeastern University, Boston, MA, USA

Ben Greenman  

University of Utah, Salt Lake City, UT, USA

Abstract

Type systems evolve too slowly to keep up with the quick evolution of libraries – especially libraries that introduce abstractions. Type tailoring offers a lightweight solution by equipping the core language with an API for modifying the elaboration of surface code into the internal language of the typechecker. Through user-programmable elaboration, tailoring rules appear to improve the precision and expressiveness of the underlying type system. Furthermore, type tailoring cooperates with the host type system by expanding to code that the host then typechecks. In the context of a hygienic metaprogramming system, tailoring rules can even harmoniously compose with one another.

Type tailoring has emerged as a theme across several languages and metaprogramming systems, but never with direct support and rarely in the same shape twice. For example, both OCaml and Typed Racket enable forms of tailoring, but in quite different ways. This paper identifies key dimensions of type tailoring systems and tradeoffs along each dimension. It demonstrates the usefulness of tailoring with examples that cover sized vectors, database queries, and optional types. Finally, it outlines a vision for future research at the intersection of types and metaprogramming.

2012 ACM Subject Classification Software and its engineering → Extensible languages

Keywords and phrases Types, Metaprogramming, Macros, Partial Evaluation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.44

Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact):*

<https://doi.org/10.4230/DARTS.10.2.24>

Software (Artifact): <https://doi.org/10.5281/zenodo.12726060> [108]

Funding This work was partially supported by DOE Office of Science Contract DE-SC0022252, XStack, ComPort, “Rigorous Testing Methods to Safeguard Software Porting” and by NSF grants SHF 1518844, CCF 2217154, and CCF/CSE 2030859 to the CRA for the CIFellows project. Felleisen’s research was partially supported by several NSF grants (SHF 2007686, 2116372, 2315884).

Acknowledgements Thanks to Sam Tobin-Hochstadt for inspiring tailoring in Typed Racket, to Mark Erickson for teaching best practices of Phoenix Verified Routes, to Ryan Culpepper for syntax parse support, to Matthew Flatt for help with Rhombus annotations, and to Alex Knauth, Asumu Takikawa, Gabriel Scherer, Justin Slepak, Leif Andersen, Scott Wiersdorf, and Zeina Migeed for comments on early drafts.

1 Type Tailoring Helps Programmers

Every typed language should come with a *type tailoring* toolkit to let programmers systematically rewrite code before it reaches the typechecker. Tailoring is essential for keeping up with libraries, domain specific languages, and even built-in embedded languages. Consider

 © Ashton Wiersdorf, Stephen Chang, Matthias Felleisen, and Ben Greenman;
licensed under Creative Commons License CC-BY 4.0
38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 44; pp. 44:1–44:27



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

44.2 Type Tailoring

regular expressions, which are often embedded in strings. The following example builds a regular expression that matches and extracts a two-digit number from a username. Without tailoring, the Typed Racket typechecker rejects this program:

```
(define dig-pat "[0-9]")
(define two-digs (string-append dig-pat dig-pat))

(define (user-idnum (username : String)) : Number
  (define full-pat (string-append "(" two-digs ")"))
  (define m (regexp-match full-pat username))
  (if m
      (string->number (second m))
      (error "bad username")))

(user-idnum "dent42")
```

Language: Typed Racket

Without Tailoring

Type error: second
argument: (Listof (Option String))
expected result: String

With Tailoring

Success: 42

Programmers fluent in regular expressions know that a call to `user-idnum` returns a number when `regexp-match` succeeds; otherwise, `user-idnum` raises an error. A standard type system knows much less because it ignores the values of strings: it does not know that the first capture group (`second m`) must exist when the match succeeds, and it does not know that the call to `string->number` must also succeed. To resolve these issues, the typechecker requires explicit checks and casts. Type tailoring can insert sufficient casts automatically by analyzing the regular expression string and transforming the program, thereby convincing the typechecker that the code is safe. For programmers, the net result is that the code above just works – without the clutter of casts, and without the need to migrate to an alternative regular expression syntax [58, 95, 107].

Type tailoring improves type analysis by observing and propagating static information. First, a tailoring for regular expressions observes the structure of the pattern string and finds that it has balanced parentheses that enclose a two-digit pattern:

```
(define full-pat
  (string-append "(" two-digs ")"))
```

Tailoring sees: full-pat is a literal:
"([0-9] [0-9])".

If the string were not available for static analysis there would be nothing to observe:

```
(define hidden-pat (read-line))
```

Tailoring sees: hidden-pat is a string.

Second, tailoring propagates information about the pattern string from where it is defined to where it is used. Since this example is implemented in Typed Racket, it uses Racket syntax properties to store and retrieve metadata; other languages use similar mechanisms (Section 3). At the `regexp-match` call, this static information implies that the result of a successful match must be a specific list value:

```
(regexp-match full-pat username)
```

Tailoring sees: since full-pat has one capture group, the match returns either #false or a list of two strings, the second of which consists of two digits.

Third, tailoring elaborates (rewrites) the `regexp-match` call to cast a successful match result, thereby communicating the structure of the result value to the typechecker. Similarly, tailoring can elaborate the access (`second m`) to a faster, unsafe memory access because it is sure to be in bounds when the match succeeds:

```
(if m (second m) ...)
  ~~ (if m (unsafe-ref m 1) ...)
```

Tailoring sees: since `m` is a list in this branch, it has two strings.

Information about the extracted string flows into a tailoring for `string->number` and justifies a final cast to convince the typechecker that the result must be number.

```
(string->number •)
  ~~ (cast (string->number •) Number)
```

Tailoring sees: the cast cannot fail because `•` evaluates to a two-digit string.

Stepping back from this example, the overall message is that incorporating a bit of partial evaluation, flow analysis, and metaprogramming into the front end of a conventional typechecker is an effective way to support domain-specific typing for embedded languages. Type tailorings can compose with one another and can enhance an entire module with a few changes to its preamble (e.g. by importing a tailored regular expression library) rather than whole-program edits. There is of course a risk that arbitrary tailorings can perform unexpected or unsafe elaborations, but we show in Section 5 how the authors of tailorings can mitigate these concerns by appealing to baseline program behavior and static information.

Contributions

Type tailoring has appeared in many contexts, but never as an officially-supported language feature. This paper analyzes the spectrum of type tailoring across languages and libraries, identifies the linguistic features that make tailoring work, and establishes a framework for future research to push the boundaries of *end-user* programmable type elaboration. Concretely, the paper makes the following contributions:

- it proposes *type tailoring* as an overarching concept in user-level metaprogramming that should be recognized and more-widely adopted;
- it demonstrates the usefulness of tailoring with a variety of examples using Typed Racket, Rhombus, Julia, and Elixir (Section 2);
- it analyzes tailoring systems along six technical dimensions (Section 3), thereby revealing three notable points in the language design space (Section 4); and
- it provides a recipe for reasoning about the behavior of tailorings and establishing the validity of their transformations (Section 5).

The paper concludes with related work (Section 6), future work (Section 7), and takeaways (Section 8).

2 Tailoring in Action

To illustrate the variety of type tailoring, this section presents five examples in four languages: a tailoring framework and type programming in Typed Racket [102], dynamic typing in Static Rhombus [33], sized arrays in Julia [7], and statically-checked web routes in Elixir [28]. Each example contributes a unique perspective to showcase the range of type tailoring applications. Typed Racket supports a family of related tailorings, Rhombus weakens types instead of strengthening them, Julia achieves order-of-magnitude performance improvements, and Elixir shows how tailoring can benefit an untyped language.

2.1 Refining Data in Typed Racket

Typed Racket enables type tailoring by exposing key pieces of Racket’s macro API and by typechecking code after macro expansion [21, 104]. Macros can thus inspect and transform code to manipulate what the typechecker sees. The `trivial` library [40, 41] uses this API to tailor a variety of domains from `printf` strings to SQL queries and beyond.

44:4 Type Tailoring

Format Strings

For `printf`, tailoring uncovers static information from escape characters in format strings. This tailoring stores information in a dictionary, mapping the key `fmt-args` to the expected types of the remaining arguments of `printf`. (Through the use of a key-value store, multiple tailorings can work together, as we will see later in this section.)

```
(require trivial)          Typed Racket
(define fmt1 "hello ~a\n") :: { fmt-args : [any] }
(define fmt2 "int to bin: ~b\n") :: { fmt-args : [Integer] }
```

Calls to `printf` statically check whether their first argument has format information. When this information is present, the tailoring checks the number and type of other arguments and raises an error at compile time if there is a mismatch. Without tailoring, such checks do not happen until runtime:

```
(printf fmt1 "world")
(printf fmt1 "john" "hancock")
(printf fmt2 "NaN")
```

Typed Racket

Without Tailoring

- Runtime errors: `printf`
 - `[fmt1]` expected 1 argument, given 2
 - `[fmt2]` expected an integer, given something else

With Tailoring

- Tailoring errors:
 - `[fmt1]` expected 1 argument, given 2
 - `[fmt2]` expected Integer, given String

Query Strings

For SQL queries, two sources of information come together to provide static checks via tailoring: database schemas and query strings. Programmers must write the schemas as type annotations in a notation specified by the SQL tailoring. Query strings use conventional SQL syntax to access the database. The tailoring parses query strings to reveal type constraints.

In the following example, the schema argument states that the database has one table named `Cats` with three columns for an identifier, pet name, and breed. Tailoring elaborates the `query-row` call to validate argument types. Without tailoring, the database executes the nonsensical query and returns an empty result:

```
(define db
  (sqlite3-connect #:user "user"
                  #:database "Pets"
                  #:schema [Cats
                            [(id : Integer)
                             (name : String)
                             (breed : String)]]))

(query-row db
           "SELECT breed FROM Cats
            WHERE name = ?"
           69105)
```

Typed Racket

Without Tailoring

- Runtime error: `query-row`
query returned zero rows

With Tailoring

- Tailoring error:
expected String, given Integer

This tailoring additionally propagates information about the result of a query. When a query selects only the `breed` column, the result is a vector with only one string value:

```
(query-row db          Typed Racket
           "SELECT breed FROM Cats
            WHERE name = ?"
           "mittens") :: { type : (Vector String) }
```

While the `regexp-match` example from Section 1 and the `printf` example from this section improve code with no effort from users, the SQL tailoring cannot act without a schema as input. But, since the tailoring works through surface syntax, it can get this input in an idiomatic way without being constrained by the typechecker or host language. In particular, tailoring adds support for the `#:schema` argument by parsing the schema and elaborating to a plain `sqlite3-connect` call with only two keyword arguments.

Cooperating Tailorings

Static information embedded in strings can be useful in domains beyond `printf`, database queries, and regular expressions. By storing domain-specific information in a dictionary and using uniquely generated keys (Section 3.4), different tailorings can annotate the same value. For example, the following string has at least three interesting properties:

```
(define str          Typed Racket
  "(SELECT breed FROM Cats)") :: { rx-groups : 1
                                     db       : [SELECT (breed) Cats]
                                     string-len : 24 }
```

Of course, strings are not the only data structure that get repurposed in domain-specific ways. Vectors, lists, functions, and numbers also benefit from tailoring:

```
(define buffer        Typed Racket
  (make-vector (expt 2 5))) :: { vector-len : 32 }

(define (swap ab)
  (list (second ab) (first ab))) :: { fn-arity : 1 }

(define pairs '((1 2) (3 4)))
  :: { list-len : 2 }
(map swap pairs) :: { list-len : 2 }
```

In all cases, there is a general recipe at hand:

- Static information originates in surface syntax, such as the characters in a string literal or the shape of a function declaration. Information can also come from an external source, such as a database schema or online API specification.
- When static information is present, type tailoring attaches it to variables and propagates it through operations such as `map` and `regexp-match`.
- Tailoring elaborates surface syntax to code that the host typechecker can understand.

Defining a new tailoring requires three steps. First, define a unique key (e.g. via `gensym`). Second, create tailored variants of constructors that identify and attach static information. Third, create wrapper macros (i.e., compile-time functions) around various operations to leverage static information when it is present and otherwise preserve the default behavior.

Evaluation: Typing Regular Expressions

Regular expression tailoring comes with immediate benefits for typed code because, by default, programmers must use casts to guard against match failures even when such failures obviously cannot occur. In Typed Racket, the need for casts arises from the conservative

44:6 Type Tailoring

type of `regexp-match` results, which says that all match results are either `#false` or lists with one string element and an unknown number of additional elements that are either strings or `#false`. This type is always correct but usually too imprecise to be useful:

```
(regexp-match full-pat username)
: (Option (Pairof String (Listof (Option String))))
```

Typed Racket

Searching the Racket 6.5 distribution and code on its package server revealed 160 files using `regexp-match` with capture groups. Migrating the files to use type tailoring obviated the need for casts in 116 originally-untyped files and 6 typed files. These improvements resolved a total of 329 false type errors (that Typed Racket would have reported) across 93 % of all `regexp-match` occurrences in the dataset [108].

Only 38 files were not improved by tailoring. Most of these files (20 of 38) extracted a capture group, but did not depend on the result being a string. The others either used non-constant pattern strings (5 files), used helper functions to assemble patterns (9 files), or used patterns with groups that may indeed fail to capture – such as "(a)|(b)" (4 files).

Evaluation: Predicting Vector Bounds

Racket library code occasionally employs fixed-size vectors. For example, the built-in `gzip` implementation declares vector constants to implement a Huffman tree. When these vectors get accessed with a statically-known index, tailoring can refine code to skip the bounds check.

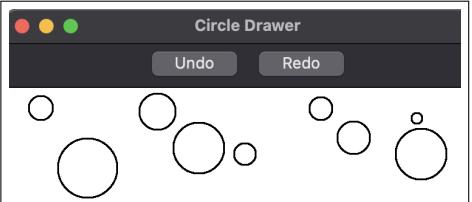
Across the Racket core distribution and packages, we found 88 files using vector constants. Tailoring eliminated bounds checks in 11 files. This number is low, but within these few files, tailoring improved 104 bounds checks in total. Most of these (80 of 104) appear in a Parcheesi implementation that uses a macro to generate code that accesses valid locations; this example shows that tailorings are robust even when other metaprogramming is present.

2.2 Elaborating Types in Typed Racket

Types themselves can benefit from tailoring. Since types are mere syntax before the type-checker gives them meaning, tailoring can elaborate declarative syntax into fine-tuned types.

Concise GUI Subtypes

Felleisen's implementation [29] of the 7GUI benchmark [53, 54] uses tailoring to simplify class type declarations. In Typed Racket, subclasses must declare types for all inherited methods and fields. While this requirement means that a subclass may refine the types of methods defined in its superclass, it also imposes a significant burden on programmers. In 7GUI, subclasses of the `Canvas%` type would normally have to spell out the types of thirteen methods. Type tailoring lets programmers specify just the differences from the parent class:



```
(define-type-canvas Circle-Canvas%
 #:minus-init (paint-callback)
 (unlock (-> Void))
 (draw-circles
 (-> (Optional Circle)
 (Optional (Listof Circle))
 Void)))
```

TR

Without Tailoring	With Tailoring
13 method declarations (not shown)	2 method declarations, 1 subtraction

The tailored 7GUI specification for a circle-drawing canvas simply gives the name of one constructor input to remove (`paint-callback`) and the types for two methods to add: an `unlock` method to freeze the canvas (`lock` is private) and a method to draw circles on the canvas. Without tailoring, this specification would require twelve redundant names and types from the parent class.

Functional-Style Object Types

The `zombie` program from the GTP benchmark suite [42] presents an example of types tailored for readability. The original, untyped program uses functions to mimic message-passing objects [106]. Equipping such a program with types is challenging. For example, the function `new-zombie` takes a coordinate pair (`Posn`) and returns another function (`Zombie`) from a symbol to a method representation. These method representations are pairs that combine a label and function. In the code below, there are two methods that have different types:

```
(define (new-zombie p) ;; Posn -> Zombie
  (λ (msg)
    (case msg
      [(can-grab?)
       (cons 'can-grab? ;; Method 1: Posn -> Bool
             (λ (q) (<= ((posn-dist p) q) *grab-radius*)))
      [(move-to)
       (cons 'move-to ;; Method 2: Posn -> Zombie
             (λ (q) (new-zombie ((posn-move-to p) q *speed*))))]
      [else
       (error "unknown message")])))
```

Typed Racket

Although Typed Racket can express the overloaded return type for a `Zombie` object (via singleton types for labels [44]), writing such types requires intimate knowledge of the encoding. With tailoring, the types can essentially match Typed Racket's object type syntax:

```
(define-type Zombie
  (-> Symbol
    (U (Pair 'can-grab? (-> Posn Bool))
        (Pair 'move-to (-> Posn Zombie)))))
```

Typed Racket

```
(define-obj-ty Zombie
  [can-grab? (-> Posn Bool)]
  [move-to (-> Posn Zombie)])
```

TR

Without Tailoring

Encoding with pair and union types

With Tailoring

Domain-specific representation

Type Expanders

The type expanders library [94] can build types such as `Zombie` and `Circle-Canvas%` *within* another type, without the need to declare a top level definition via `define-type`. The following example, from the library documentation, presents a type expander `HomogeneousList` that uses an integer literal to expand to a `List` type:

```
(: five-strings (-> String (HomogeneousList String 5)))
(define (five-strings x)
  (list x "a" "b" "c" "d"))
```

Typed Racket

Without Tailoring

`(List String String String String String)`

With Tailoring

`(HomogeneousList String 5)`

Type expanders supports a wide range of additional tailorings. These include type abstraction (Λ) and local definitions in types (`Let`).

2.3 Relaxing Types in Rhombus

Rhombus enables tailoring in the same manner as Typed Racket by inheriting Racket's metaprogramming tools [33]. To illustrate, we equip the static variant of Rhombus (akin to strict JavaScript [25]) with a dynamic type (`Dyn`) in the spirit of optional and gradual typing [92, 100, 101]. Gradually typed languages are often defined by elaboration into a typed language with casts, making them a natural application for tailoring.

Rhombus comes with an annotation language that can, among other things, statically resolve method calls. For example, a function whose argument has the `List` annotation knows where to find the appropriate `length` method for this argument:

```
fun len_plus_one(l :: List):
    l.length() + 1
```

Static Rhombus

Static Rhombus requires annotations for the receivers of all methods calls, all list and map lookups, and similar operations [80]. It uses these annotations to guard against errors and to compile optimized code. Getting the annotations right can become a burden, as the long history of gradual typing attests [39, 68, 93, 103]. It would be useful to selectively disable the requirement, but by default Rhombus provides only a coarse-grained solution via a keyword `use_dynamic` that disables annotation requirements within an entire block of code.

The `Dyn` tailoring is an annotation that selectively disables static checks for an individual variable without losing guarantees in other parts of the same code block. This can be useful, for example, to implement a dynamic equality function for lists:

```
fun list_equals(l1 :: List.of(Dyn), l2 :: List.of(Dyn)):
    def len = l1.length()
    len == l2.length()
    && for all (i: 0..len):
        l1[i].equals(l2[i])
```

Static Rhombus

Without Tailoring

Compile error: `l1[i].equals`
`no such method based on static information`

With Tailoring

Success

Static Rhombus accepts this code and resolves the call to `.equals()` at runtime. Other calls, such as `l1.length()` and the `for`-comprehension iterator, resolve statically.

Evaluation: Using Dyn in Shplait

Shplait is a typed, ML-like language developed for teaching and implemented in Rhombus [33]. It is one of the largest Rhombus programs to date. Within the Shplait codebase, there are 84 type annotations that appear in function, variable, and class definitions. These annotations appear in 21 of the 46 core Shplait files. Replacing these annotations with `Dyn` does not raise any compilation errors. The only difficulties that arose were due to import clashes with operations that `Dyn` overrides (such as `++`), which were straightforward to resolve by importing `Dyn` with a prefix.

2.4 Static Arrays in Julia

Array accesses in Julia incur a runtime bounds check by default. This check can become a significant and unnecessary cost in scientific code. For example, the coordinates for bodies in an N -body simulation might be stored in fixed-length arrays that get accessed in a highly repetitive pattern. A bounds check on each access would quickly incur a significant and unnecessary performance cost.

The StaticArrays package implements a form of type tailoring that elaborates normal-looking array code into fixed-size tuples [97]. To declare a static array, programmers wrap a normal array declaration in the `@SVector` macro. The package supports several kinds of declarations, including array comprehensions:

<code>vec1 = @SVector [1]</code>	<code>Julia</code>	:: { <i>vector-len</i> : 1 }
<code>vec3 = @SVector zeros(3)</code>	<code>Julia</code>	:: { <i>vector-len</i> : 3 }
<code>vec9 = @SVector [i^2 for i = 1:9]</code>	<code>Julia</code>	:: { <i>vector-len</i> : 9 }

In addition to array constructors and references, StaticArrays tailors a variety of linear algebra operations to use size information. Matrix multiplication, transposition, and reshaping can all propagate sizes. Eigenvalue decomposition uses a fast algorithm for small matrices:

<code>m3 = @SMatrix randn(3,3)</code>	<code>Julia</code>	:: { <i>matrix-shape</i> : (3, 3) }
<code>eigen(m3).values</code>	<code>Julia</code>	:: { <i>vector-len</i> : 3 }

Elaborating arrays to tuples is impractical for large arrays in Julia; the StaticArrays documentation recommends 100 elements as a rule-of-thumb upper bound. Nevertheless, StaticArrays is an important part of the Julia ecosystem. As of January 2024, it has over 800 direct dependents and 3,000 indirect dependents (30 % of the 10,292 packages on JuliaHub). One of its clients is the popular OrdinaryDiffEq package [89], which helps explain the large number of indirect dependents.

Evaluation: Fast Matrix Rotations

The documentation for StaticArrays reports order-of-magnitude speedups on a variety of linear algebra microbenchmarks using 3×3 matrices [97]. Examples include a 5.9x speedup for matrix multiplication, a 113x speedup for determinant computation, and a 8.8x speedup for Cholesky decomposition. We built our own microbenchmark that rotates a vector by a 10×10 matrix one hundred million times. The results in Table 1 show a 10x speedup and a dramatic decrease in memory use thanks to inlining and predictable array layout. These numbers are the average after two hours of sampling with `BenchmarkTools.jl` in Julia 1.8 on a single-user Linux machine with 4 physical i7-4790 3.60GHz cores and 16GB RAM.

Table 1 StaticArrays yields a 10x speedup on a synthetic matrix rotation benchmark.

	mean (stddev)	Memory Use	GC % (stddev)	samples
Without Tailoring: Arrays	10.7 s (8.3 ms)	13 GB	1.7% (0.07%)	671
With Tailoring: StaticArrays	1.5 s (8.9 µs)	0 B	0% (0%)	4856

2.5 Verified Web Routes in Elixir

The Phoenix web framework [76] uses Elixir’s macro system to validate web routes. Programmers declare routes and corresponding handlers in a dedicated module. Phoenix leverages information from the route module in a three-step validation process: first, it collects route references that marked by a certain macro (`~p`); second, it elaborates these marked references into plain strings; and third, it checks that each reference has a matching handler.

Route references commonly appear in page templates. For example, the following template code block contains the references `/users/register` and `/users/login`. Suppose that the second route has a typo, and the correct path is `/users/log_in` with an underscore. Without tailoring, routes are mere strings; a typo in a route name is not a problem until a user requests the page and gets a 404 error. With tailoring, a static check catches the error immediately.

```
<p>
  <%= link "Register", to: ~p"/users/register" %>
  <%= link "Log in", to: ~p"/users/login" %>
</p>
```

Elixir

Without Tailoring

Possible 404 at runtime

With TailoringTailoring error:
no route path matches /users/login

Thus, even a dynamically typed language such as Elixir can benefit from domain-specific static checks during the elaboration of source code to baseline Elixir. Prior work in Scheme illustrates the same point in several other domains [45], though at a much smaller scale than Phoenix. Ruby on Rails [6] and Haskell [66] have their own methods of static route validation; this is a common issue that tailoring helps to solve.

Evaluation: Adoption Data

Phoenix introduced verified routes in February 2023 [65]. They are an optional feature for existing projects, while for new projects Phoenix generates code with verified routes by default. As of January 2024, over 1,800 Elixir files on GitHub are using verified routes [36].

3 Dimensions of Tailoring Systems

Many languages and libraries support a form of type tailoring. Examples include the macro systems in Clojure [18], Scala 2 [9], Scala 3 [87], and Rust [86]; elaborators in Idris 1 [13]; Template Haskell [90]; OCaml PPX [70], MetaOCaml [52], and MacoCaml [110]; CompRDL [50]; and type providers in the style of F# [14, 75]. Despite differences in their specific aims and affordances, they all enable metaprogramming of the elaboration from surface code into typechecked code.

As a step toward an analysis of language support for tailoring and of relative strengths and weaknesses, this section introduces a framework of *technical dimensions* (inspired by prior work [38, 46]) for type tailoring. The dimensions fall into two groups: the first four describe metaprogramming features, and the last two describe contextual information that may be available to tailoring:

Metaprogramming Features

Metadata. For tailoring to work together, they must have a way of sharing static information. Metaprogramming systems that can attach metadata directly to AST nodes enable this sharing in a direct way. (Compile-time state is an alternative.)

Binding. How to handle bindings is a crucial aspect of information sharing. Information must be able to flow from a variable declaration to its use. Metaprogramming systems can discover these connections and expose them to tailoring.

Order. Cooperating tailoring need a reliable and customizable order of expansion. One tailoring might depend on input from another, and it may wish to have a third tailoring analyze its output.

Hygiene. To a first approximation, a hygienic metaprogramming system respects the lexical structure of code. Hygiene is important for tailoring to compose with one another and with user code. Users, for example, should not need to worry about whether the `f` in `f(x)` is a tailoring (e.g., a macro) or a normal function.

■ **Table 2** Technical dimensions of tailoring systems.

System	Metaprogramming				Context	
	Metadata	Binding	Order	Hygiene	Definitions	Types
Racket	●	●	●	●	●	×
Clojure	●	●	●	○	●	×
Elixir	●	✗	●	●	●	✗
Julia	○	✗	●	○	●	✗
Idris 1	✗	○	○	✗	●	●
Scala 3	✗	✗	●	●	●	●
Template Haskell	✗	✗	○	●	○	✗
Type Providers	✗	✗	✗	✗	●	○
OCaml PPX	○	✗	✗	✗	○	✗
Rust	✗	✗	○	○	●	✗

● Full support ○ Partial support ✗ No support

Context Information

Definitions. Tailorings benefit from local definitions and external data, such as a database, as sources of static information. Without definitions, tailorings are limited to local transformations such as refining calls to `printf` that apply a literal format string.

Types. Type context is a dimension that has benefits and drawbacks. On one hand, if tailorings receive typechecked input then they can leverage the types and need not handle malformed syntax. On the other hand, types restrict the shape of domain-specific syntax to terms of the host language.

Table 2 presents an evaluation of representative tailoring systems along these technical dimensions. The rows are *not* an exhaustive list of tailoring systems but rather give an overview of distinct feature-sets. For example, there is no row for Rhombus because it has the same feature set as Racket. The table suggests two high-level takeaways:

1. A number of systems lack support along several dimensions. These represent trade-offs that realize some benefits at a modest effort. Section 4 examines three points in depth.
2. No system has full support along every dimension. Section 7 presents ideas for designing a full-featured system as future work.

The rest of this section explores the cells of Table 2 in detail. For each dimension, a subsection provides a detailed description of what it means and justifies the partial cells (○). The last subsection gives a concrete implementation of a tailoring for static vector references; this example shows how one tailoring benefits from several dimensions.

3.1 Metadata

Static information is metadata. Examples include the length of a string literal, the capture groups in a regular expression, and the schema of a database. Tailorings discover this metadata, and they need a way to disseminate it to reap the benefits. For example, discovering the length of a literal vector may be useful by itself, but it is more useful if length information can propagate through operations such as vector concatenation.

Attaching metadata to AST nodes is a direct way to propagate information. Any piece of syntax – whether it describes a value, an expression, or a definition – should support metadata. Furthermore, as the examples from Section 2.1 demonstrate, the metadata should

be a key-value store that can hold data structures as values. Keys clarify the interpretation of data such as numbers, which, in the examples, represent lengths and regexp groupings. Structured values declaratively express format-string constraints and database schemas.

Racket [78], Clojure [19], Rhombus [81], and Elixir [27] support general key-value metadata on arbitrary nodes. Julia [48] and OCaml [70] support a limited form of metadata; only a specific type of AST node can hold metadata. These metadata nodes can, however, be inserted as siblings to other nodes in the syntax tree. To the best of our knowledge, the other systems in Table 2 have no direct support for metadata, though Idris has highly-expressive dependent types that can achieve similar goals.

3.2 Binding

Variable declarations call for a special kind of metadata that flows from a binding to its references. Consider a basic `let` expression that binds a format string to a variable `str`; a tailoring system should ensure that calls to `printf` within the body have access to format metadata:

```
(let ([str "age: ~a"])
  ...
  (printf str n) ;; need data here
  ...
  (lambda (str) (printf str n))) ;; but not here
```

Racket

In Racket and Rhombus, *rename transformers* flow data to references [79]. Clojure has libraries for similar functionality [17, 64]. None of the other systems support renaming in a programmatic manner, which means that the authors of tailorings need to manage propagation on their own – perhaps by handcrafting AST structures.

3.3 Order

Control over the order of elaboration makes it possible for tailorings to share their results with one another. For example, static-length vectors and constant folding are somewhat useful in their own right, but are more effective together:

```
(define buf-size (* 4 4))      Typed Racket :: { int-value : 16 }
(define buffer (make-vector buf-size)) :: { vector-len : 16 }
(sub1 (vector-length buffer))    :: { int-value : 15 }
```

Unlike in a normal metaprogramming system, it is crucial that the order of elaboration matches the order of runtime evaluation. The tailoring for `make-vector` must happen after the tailoring for multiplication, and the tailoring for `sub1` must happen last of all because it depends on the first two results.

Clojure macros give control over ordering in a simple way through a `macroexpand` directive [69] inherited from Lisp [98] and Scheme [23]. Julia [49] and Elixir [27] provide a similar directive. This method is unhygienic, and may cause problems when macros depend on one another [31]. There are several alternative forms of sequencing that fall short of arbitrary ordering. Elixir provides compile-time hooks to register code that runs before and after a module compiles; these hooks let Phoenix (Section 2.5) verify routes after registration. Template Haskell allows stacks of tailorings, but programmers must manage them explicitly by wrapping each piece of syntax in a suitable number of template quotes [90]; the Haskell type system does help to manage the layers. Idris elaborators require similar management [13]. Rust macros can expand to other macros that have been defined in a separate crate [86]. Type providers in F# cannot expand to one another [75]. OCaml PPX recommends that users do not rely on the order of expansion [70].

3.4 Hygiene

A hygienic metaprogramming system enables composable tailorings. Macro hygiene ensures that compile-time transformations respect the binding structure of the code they manipulate. A classic illustration is an `or` macro that introduces a temporary variable:

```
(defmacro or (a b)
  '(let ((tmp ,a))
    (if tmp tmp ,b)))
```

Common Lisp

If a call to this macro simply replaces code, the `tmp` variable can shadow a binding and produce the wrong result:

```
; ; before expansion
(let ((tmp 42))
  (or nil tmp))
;; expected result: 42
```

```
; ; after unhygienic expansion
(let ((tmp 42))
  (let ((tmp nil))
    (if tmp tmp tmp)))
;; actual result: nil
```

A second hygiene issue concerns references from macro definitions to functions. For example, the SQL tailoring from Section 2.1 relies on helper functions to analyze strings. If the helpers’ names were to get shadowed at the macro use-site – as is the case with unhygienic systems – the tailoring would crash or produce flawed results.

A third related issue is that tailorings ought to work as a drop-in replacement in user code. Code that calls a standard function such as `make-vector` should work with a tailored variant instead, without the programmer needing to annotate the call site as a macro call rather than a function call. Since functions are typically first-class values, this means that macros must work hygienically in first-class use-sites.

Lastly, the keys used to label static information need a form of hygiene. If two tailorings inadvertently choose the same key, they may attach conflicting information to a value. Tailoring systems must provide a facility to generate unique keys – such as `gensym` in Julia and other languages – to prevent clashes.

Racket [32], Rhombus [33], Elixir [27], Scala 3 [88], Template Haskell [90] all support macro hygiene. Julia is hygienic for simple macros, but in complex macros programmers must manually rename variables to avoid issues [49]. Rust’s support for hygienic macros is currently experimental [51, 86]. F# type providers, OCaml PPX [70], Idris [13], and Scala 2 [9] provide no support for hygienic macros.

3.5 Definitions in Context

There are two aspects of definitions that relate to tailoring. The first and most important is access to external data, whether it be a relational database (Section 2.1), a manifest of web routes (Section 2.5), or a JSON endpoint [75]. To add two more examples to the mix, CompRDL uses domain-specific knowledge of Ruby on Rails and database schemas to achieve dependent types for table functions [50], and the Rust SQLx library checks the well-formedness of query strings [59]. Both leverage external data to analyze code without asking programmers to change their idiomatic code (conventional Rails in Ruby, raw SQL in Rust). Every tailoring system in Table 2 provides access to external data.

The second aspect of definitions is access to local variables and helper functions. As mentioned in Section 3.4, a tailoring that parses query strings benefits from access to helper functions. (Without access, the tailoring must duplicate code in its definition, which then increases the size of the object code.) Most systems provide access to local definitions, though

in the context of unhygienic systems this must be done with care. Template Haskell code can access local definitions but is subject to Haskell’s disciplined use of side effects [90]. OCaml PPX runs macros in isolation, so they cannot use compile-time definitions [70].

3.6 Types in Context

In Scala, the typechecker validates input to macros as well as the expanded code. Types provide extra context to tailorings and detect certain malformed input. However, types also put restrictions on inputs. In the Squid metaprogramming framework for Scala 2 [74], programmers typically wrap code in quasiquotes to bypass the surface typechecker. The following example is from the Squid documentation [73]:

```
val powCode = code"${(x: Variable[Double]) => mkPow(code"$x", Const(n))}"
```

After some syntactic adaptations, the expanded code is thoroughly typechecked. (Squid also accepts Scala code fragments directly; these inputs must be well-typed.)

Idris distinguishes between typed and raw terms during elaboration [13]. All terms must eventually pass the typechecker, but elaboration can manipulate both sorts of terms. The other systems in Table 2 do not typecheck their input, though type providers can effectively rely on types because their inputs are restricted to literal constants. There are, however, several metaprogramming systems in the literature that do typecheck their inputs. Examples include SoundX (which furthermore guarantees well-typed transformations) [62], Wyvern [71], Dependent ML [109], and Scala 2 [9].

3.7 Essential Non-Dimensions

Table 2 focuses on elements that are significant for tailoring but lack full adoption. As such, it omits basic features that enable type tailoring. The following are requirements rather than dimensions, but they are nevertheless important for language designers to know about:

Typechecking After Elaboration Regardless of whether or not typechecking happens before elaboration, it must happen afterward to check the results of user-defined tailorings.

Elaboration-Time Computation Tailorings need infrastructure, such as procedural macros, to perform non-trivial computations. By contrast, pattern-based macros (which unpack the syntax of their call-site and rearrange it [56]) cannot even read from an external file.

AST Datatype Without an AST datatype, tailorings are limited to using a token stream as input and output (e.g., in Rust [86]). Token streams cannot carry metadata (though it might be stored off to the side) and must be parsed to find their binding structure.

3.8 Example Tailoring Implementation

Let us show how the dimensions work together with an example that tailors vector references in Racket. This demo replaces `vector-ref` with either a fast `unsafe-vector-ref` or an error when it finds both a literal vector and a literal index; otherwise, it leaves the reference as is:

```
(vector-ref (vector 5 2 8) 1)  ~>  (unsafe-vector-ref (vector 5 2 8) 1)
(vector-ref (vector 4 9 1) 4)  ~>  error: out-of-bounds
(vector-ref (read-vec) 9)     ~>  (vector-ref (read-vec) 9)
```

See the documentation of the `trivial` library [41] for a full-featured implementation that works for identifiers as well as data literals.

The six code blocks below define the `vector-ref` tailoring. When defining the tailoring, we use the name `tailored-vector-ref` to avoid shadowing the base `vector-ref` function. On export, we rename this tailoring to `vector-ref` so clients of this library can use the tailored version as a drop-in replacement in their code:

```
(provide
  (rename-out [tailored-vector-ref vector-ref]))
```

Racket, 1/6

The tailoring module imports `unsafe-vector-ref` for use in expanded code, and three other libraries to define the tailoring:

```
(require
  (only-in racket/unsafe/ops unsafe-vector-ref)
  (for-syntax ;; import these things for macros
    racket/base syntax/parse (only-in "tailoring-api.rkt" ~φ V I)))
```

Racket, 2/6

The helper module `tailoring-api.rkt` is a small wrapper over Racket's metaprogramming facilities. Specifically, it provides a bridge for the following dimensions:

Order of expansion (\rightsquigarrow) The syntax class \rightsquigarrow triggers macro expansion on a subexpression, allowing the tailoring to discover static information.

Metadata (ϕ) The function ϕ uses Racket syntax properties to store and retrieve static information using domain-specific keys.

Hygiene (V, I) The keys V and I are unique gensyms for vector length and integer value information. (Under the hood, `tailoring-api.rkt` registers such information when it encounters relevant data literals during expansion.) Uniqueness means that other tailorings cannot accidentally use the same names and cause a collision; however, it does not stop a malicious tailoring from writing bad information using the keys.

For details on `tailoring-api.rkt`, refer to the artifact for this paper.

The tailoring itself is a macro so that it can statically rewrite source code. First, it parses its input syntax object (`stx`) to extract and expand two subexpressions:

```
(define-syntax (tailored-vector-ref stx)
  (syntax-parse stx
    [(_ e1:~φ e2:~φ)])
```

Racket, 3/6

The expanded form of subexpression `e1` is available as `e1.~φ`, and similarly for `e2`. The tailoring checks whether these expanded expressions have the static information that it needs; specifically, it needs a vector length (key: V) and an integer value (key: I):

```
#:do [(define n (φ (syntax e1.~φ V)))
       (define i (φ (syntax e2.~φ I)))]
#:when (and (integer? n) (integer? i))
```

Racket, 4/6

If the information is present, the tailoring checks whether the index is in bounds and expands to code that either performs a fast vector reference or raises an exception:

```
(if (and (≤ 0 i) (< i n))
    (syntax (unsafe-vector-ref e1.~φ e2.~φ))
    (syntax (error 'Index-Exn))))
```

Racket, 5/6

Otherwise, the default behavior is whatever Racket's un-tailored `vector-ref` does:

```
[(_ e1:~φ e2:~φ)
  (syntax (vector-ref e1.~φ e2.~φ)))]
```

Racket, 6/6

With that, the tailoring is complete. A Racket program can import this tailoring to replace the default `vector-ref`:

44:16 Type Tailoring

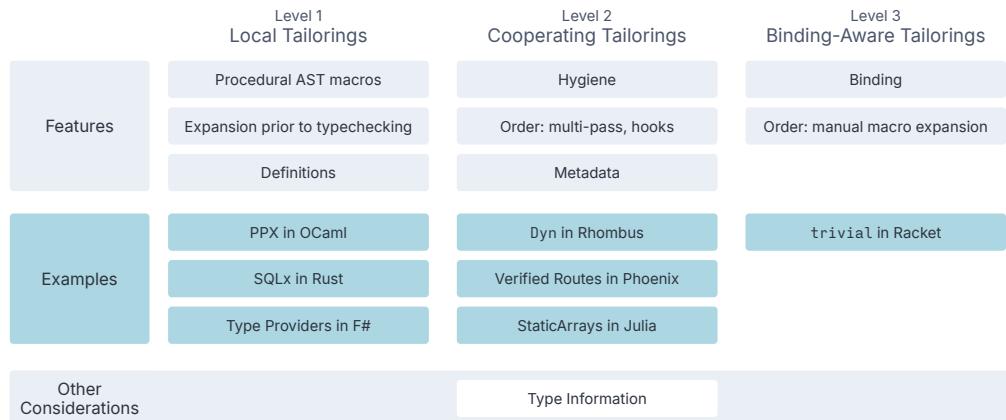


Figure 1 Three levels of support for type tailoring.

(vector-ref (vector 5 2 8) 1)

Racket

Without Tailoring

No change; use checked lookup
(vector-ref ...)

With Tailoring

Use unsafe, fast lookup
(unsafe-vector-ref ...)

In summary, tailoring `vector-ref` relies on a specific *order of expansion* to receive *metadata* about subexpressions, it extracts that metadata in a reliable way that composes with other tailorings using macro *hygiene*, and it relies on several *definitions* from the Racket standard library such as `<=` to compare numbers and Racket's `vector-ref` to provide a default behavior. Thus, four technical dimensions come together in this one example. Both the tailoring and its helper module are defined in user code; they require no changes to the Racket language to seamlessly improve client code.

4 Design Space Reflections

Tailoring systems come in wide variety in the literature, and yet they all enable at least some similarly-useful applications. For example, OCaml PPX achieves a pinch of dependent typing, F# type providers give an early warning when a web service changes its API, and Julia StaticArrays can propagate array dimensions. None of these systems can propagate metadata through binding forms, but their implementations are simpler than those that do. This tension suggests that there are points in the design space of tailorings that offer compelling tradeoffs between implementation complexity and useful capabilities.

In all, there are three important levels of tailoring support: (1) *local tailorings* that can use external data to generate types, (2) *cooperating tailorings* that share metadata and run in a customizable order, and (3) *binding-aware tailorings* that can manage an environment of static information. Figure 1 summarizes these levels both in terms of their required metaprogramming features and in terms of the tailorings these features enable. The bottom of the figure lists type information as a notable but orthogonal direction. Scala and Idris are the only systems that provide type information to macros, though it is unclear whether type information advances the frontier of tailoring systems.

4.1 Level 1: Local Tailorings

The first level of support for type tailoring enables local transformations that can query external sources of information. The PPX preprocessor in OCaml, SQLx in Rust, and type providers in F# all fall into this category. These systems can retrieve input from databases, websites, and/or constant literals to generate tailored code.

At this level, the main ingredient is support for compile-time computation: the preprocessor cannot be limited to simple pattern-and-template transformations. The preprocessor should also receive AST objects as input and it must be able to inspect these objects, query external sources, and have its output validated by the typechecker (compare to Section 3.7). Weakening any of these ingredients makes writing tailorings a challenge for language end-users, who do not have access to compiler internals. Without access to the AST, for instance, users must parse the input token stream before they can manipulate it in a meaningful way.

4.2 Level 2: Cooperating Tailorings

The second level of support allows tailorings to work together in basic ways. For example, `Dyn` in Rhombus uses metadata to tell operations how to handle an expression, `StaticArrays` in Julia lifts array structure into types, and Phoenix in Elixir collects web routes in a first pass before validating the routes in a second pass. All three require tools for sharing static information and controlling the order of elaboration.

Hygiene is crucial to enable sharing without bugs. In systems like Julia with partial support for hygiene, the authors of tailorings must fill the gap manually, for instance, by calling `gensym` to create fresh names.

Order in elaboration can come about in two ways. The direct way is to allow tailorings to expand to other tailorings. In this setting, the elaborator (macro expander) must continually process AST nodes until no tailoring uses remain. The indirect way, exemplified by Phoenix and CompRDL, is to use hooks to register tailorings that should happen at a certain point. In Phoenix, these points are just before and just after the compilation of a module.

Lastly, cooperating tailorings need a way to share information. An API for attaching metadata to AST nodes is the straightforward solution. Keeping information off to the side in metadata nodes (as in Julia) or in mutable structures is an alternative.

4.3 Level 3: Binding-Aware Tailorings

The third level of support is to equip the metaprogramming system with binding information. This level also allows fine-grained control over when the tailorings in a piece of syntax elaborate. Typed Racket requires these ingredients for its lightweight flow analysis, which lets tailorings flow through variable declarations and standard operations:

```
(let ([greeting "hello ~s"])
  (printf greeting "world"))
  ~~> (printf "hello ~s" "world")
  (make-vector (+ 40 2))
  ~~> (make-vector 42)
```

To deal with variables, tailorings need access to the binding structure of code. To ensure that subterms elaborate before the outer term, tailorings need control over the order of expansion. Among the languages in Section 3, only Racket and Clojure give full control over binding structure. Manual macro expansion is more common, with support from Elixir [27], Julia [49], and Idris [13]. Only Racket and Rhombus provide both in a hygienic way.

5 How to Reason About Tailorings

Programmers need an easy-to-use guide for the design of correct tailorings because tailorings can rewrite source code arbitrarily. Tailorings come with a degree of safety because the host language typechecks their output, but types alone do not prevent an incorrect rewrite from $\sqrt{2}$ to 42 or an unsafe rewrite that accesses out-of-bounds memory.

In general, tailorings accomplish two goals: they discover and propagate static information, and they elaborate source expressions to the host language. These goals motivate two correctness requirements. The first requirement is *prediction soundness*. If tailoring attaches static information to an expression and the expression reduces to a value, then the static information must be a correct description of the value. For example, the key-value pair `{string-len : 4}` is correct for string values with exactly 4 characters. If tailoring propagates static information for a variable, then the prediction must hold for all values the variable might take on.

The second requirement, *compatibility*, pertains to the behavior of elaborated code. Based on the examples from Section 2, there are three ways that a source expression and its tailored variant may relate to one another. Tailorings can:

- *express new behaviors* by translating invalid source syntax into valid host code;
Examples: SQL in Typed Racket (Section 2.1), Dyn in Rhombus (Section 2.3).
- *refine existing behaviors* by changing how, but not what, an expression computes; or
Example: StaticArrays in Julia (Section 2.4).
- *predict errors* by identifying a mismatch in static information.
Example: Verified Routes in Elixir (Section 2.5).

Tailorings that refine behavior or predict errors can use the untailored program as a source of truth. They should be compatible in the sense of computing equivalent values or rejecting the same programs. Tailorings that express new behavior generally require a fine-tuned correctness argument, but they may benefit from the idea of compatibility as well. For example, the Typed Racket regular expression example in Section 1 should be compatible with the baseline behavior of untyped Racket.

A Recipe

Showing that a tailoring system is correct calls for a two-part effort. First, the designers of a *cooperating* tailoring system (Section 4) must show that any propagation rules or environment-management rules respect prediction soundness. The designers of a *binding-aware* system must show that metadata propagates correctly. The designers of a *local* system have no obligations at this stage.

Second, the authors of domain-specific tailorings have three tasks:

1. Confirm the prediction soundness of rules that infer static information from values. These may be straightforward, such as inferring a length from a literal vector, and they may be sophisticated, such as the F# algorithm for JSON shapes [75].
2. Categorize tailorings that elaborate expressions as either: expressing new behavior, refining existing behavior, or predicting errors.
3. Argue that the elaborations are acceptable. Elaborations should either be compatible with some baseline behavior or desirable in some other sense.

By way of example, the next three subsections present two domain-specific tailorings and one set of general propagation rules.

5.1 Express New Behavior: Variable-Arity Map

With arity information about functions, a language with simple function types (such as Haskell, but not Typed Racket [99]) can support a variable-arity `map` function by generating code for a fixed-arity `map`. In the examples below, `map1` expects exactly one list, while `map2` expects exactly two lists and applies the function to their elements in parallel:

```
(map add1 '(1 2 3))
~~ (map1 add1 '(1 2 3))
```

```
(map max '(1 2 3) '(4 5 6))
~~ (map2 max '(1 2 3) '(4 5 6))
```

When variable-arity `map` receives a function with unknown arity, or when the arity does not match the number of lists given, it raises an exception:

```
(map max '(1 2 3))
~~ Exn: `max' expects 2 arguments, but `map' got only 1 list
```

For the first piece of static information, we need a tailoring that discovers the arity of functions – for example, that `max` takes 2 arguments:

```
(: max (-> Real Real Real))
(define (max a b)
  (if (>= a b) a b))
```

:: { fn-arity : 2 }

Prediction soundness comes from a function's arity being statically apparent. Likewise, predicting the number of list arguments is a simple matter of counting the arguments to `map`.

This tailoring is *expressing new behavior* when there is no variable-arity `map` in the source language. A reasonable baseline is to generalize the behavior of `map1`, `map2`, and so on in a compatible way. Assuming prediction soundness, `map` should elaborate to the correct `mapi` or predict the error that `mapi` would raise.

5.2 Refine Behavior and Predict Errors: Vector Bounds

When a tailoring expands to potentially-unsafe code, demonstrating soundness is crucial. Consider a tailoring that optimizes array references: when accesses (`vector-ref`) are known to be in bounds, it is safe to bypass the bounds check (`unsafe-ref`). However, if the index or the array length are not known statically, the tailoring falls back to a safe variant (`checked-ref`). This tailoring should always return the same result that a normal in-bounds access would, and should also optimize only when it is safe to do so:

```
(define my-vect (vector 1 2 3))

(vector-ref my-vect 1)           ~~ (unsafe-ref my-vect 1)
(vector-ref (read-vector) 3)     ~~ (checked-ref (read-vector) 3)
(vector-ref my-vect (read-int)) ~~ (checked-ref my-vect (read-int))
```

If this tailoring detects an out-of-bounds access statically, it can predict an error at tailoring time instead of leaving it until runtime.

```
(vector-ref my-vect 42)          ~~ Exn: Index '42' out of range
```

Information about vector sizes can come from two sources: static `vector` declarations and constructors such as `make-vector` that use statically-known sizes:

```
(vector 1 2 3)                  :: { vector-len : 3 }
(make-vector 42 0)               :: { vector-len : 42 }
```

Prediction soundness follows from the semantics of these built-ins.

This tailoring is *refining existing behavior* by optimizing accesses that are known to be safe. It also *predicts errors* when it can statically discover an out-of-bounds access. With prediction soundness in hand, behavioral soundness follows from comparing known lengths to known offsets.

5.3 Propagation and Substitution

Tailoring systems that propagate static information from variable definitions to references (i.e., *cooperating* systems in Section 4) must demonstrate *prediction soundness* for the forwarded information. Metadata must remain an accurate description for any runtime value the variable may take on. For instance, consider a vector bound to a variable `x`:

<code>(let ([x (vector 1 2 3)]) ... x ...)</code>	<code>x :: { vector-len : 3 }</code>
---	--------------------------------------

In the body of the `let`, the identifier `x` should have the information `{vector-len : 3}` attached to it. Subsequent tailorings inside the body of the `let` form should be able to take advantage of this information:

<code>(let ([x ...]) (vector-ref x 1))</code>	<code>~></code>	<code>(let ([x ...]) (unsafe-ref x 1))</code>
---	--------------------	---

At the same time, if `x` is shadowed by a different binding inside the body of the `let`, that same static information must not be attached to the new binding – transgressing lexical scoping would violate prediction soundness.

Aside from propagation through bindings, a *binding-aware* system can propagate information through a bottom-up tree traversal. For instance, `if` expressions can join the predictions from both branches:

<code>(if (daylight-savings) (vector 1 2 3) (vector 4 5 6))</code>	<code>:: { vector-len : 3 }</code>
--	------------------------------------

In this example, both branches carry the same static information; but when the branches disagree, precise information cannot propagate upward:

<code>(if (daylight-savings) (vector 1 2 3) (read-vector))</code>	<code>:: { empty map }</code>
---	-------------------------------

Exactly how to join pieces static information is context-dependent. For example, if both branches of an `if` are static vectors, but one is longer than the other, it might be sensible to propagate a length that is known to be safe. In other domains, defaulting to an empty set of properties might be the sensible thing to do.

6 Related Work

Type tailoring combines aspects of types, metaprogramming, and static analysis. In closely related work that inspired our Typed Racket tailorings, Herman and Meunier [45] show how a macro system can implement domain-specific static analyses for format strings, regular expressions, and database queries. They do not consider the interplay of domain-specific information and types. Ziggurat is a tailoring system for a C-like language [30]. It enables towers of language levels, each with a custom type system or flow analysis, and lets neighboring levels share information. Pluggable typecheckers add layers in a similar sense, though without direct support for sharing information across layers [8, 22, 67, 72]. Squid provides a framework for type-checked partial evaluation using the Scala macro system [74]. Scala LMS is another tailoring system specialized to partial evaluation that has enabled extensible optimizing compilers for domains ranging from databases to linear algebra [82, 83, 84, 85]. The finally-tagless encoding of staged interpreters is a third technique for partial evaluation that can be implemented within a typed language such as ML [10]. While its applications are more limited than the tailorings in this paper, it avoids the need for a metaprogramming layer.

The ingredients of a full-featured tailoring systems are made possible by research from the Lisp family of languages [20, 21, 33, 34, 55, 56, 98]. In particular, we note the long line of research on hygiene [1, 15, 16, 32, 55, 77].

Tailoring achieves a modicum of dependent typing in the context of a simply-typed language and without the burden of formal proof. The `printf` and vector size tailorings are similar to work on dependent types [26]. Cayenne is related as a practical compromise with dependent types; its typechecker takes care of the proof burden, but may run indefinitely [3]. Dependent type systems are common targets for metaprogramming. Prior work includes the Lean 4 macro system [24], type-directed editing and elaborator reflection in Idris [13, 58], certified metaprogramming in Coq [2], and elaboration-time solvers in Agda [57, 60]. Extensible tactic languages such as Cur [11] and VeriML [96] are tailoring systems that turn concise proofs into elementary ones.

By contrast to metaprogramming systems in general, tailoring is limited to the elaboration of surface syntax to a typed host language. Closely-related systems include Turnstile [12] and Klister [5], which create whole typed languages; Haskell typechecker plugins [4, 43], which customize the type constraint solver; and Nx [105], which compiles Elixir-like source code to GPU kernels. On a similar note, the Haskell library Servant checks web APIs statically using typechecker extensions rather than metaprogramming [66].

7 Future Work

Developing improved support for tailoring is an ongoing challenge. In addition to the quest for a system that productively combines all technical dimensions from Table 2, the following are key areas of focus going forward:

Deeper Control Flow Analysis. While the tailorings in this paper leverage some control flow analysis, this analysis is limited to local, forward propagation. No information flows through function calls, and join points (conditionals and loops) lose information to produce a conservative approximation. One way to recover precision is with annotations supported by runtime checks, though annotations put extra responsibility on programmers and checks introduce costs. Another way is to embed full [91] or demand-driven [35] control flow analysis in the tailoring system. Work on Turnstile is closely related, as it shows how macros can implement type analyses [11, 12].

Elaboration-Time Performance, How Much Metadata? The compile-time performance of tailorings has not been an issue for tailorings thus far, but it will become an issue if tailorings strive for whole-program or even whole-module analyses. Turnstile and k -CFA both suffer in this respect. A related issue is how much metadata to attach to AST nodes. Typed Racket tailoring (Section 2.1) inspects every data literal for every form of domain-specific information, which means that a string value can carry several kinds of information if it matches regular expression, `printf`, and SQL query syntax. Tracking information may, at some point, incur a noticeable compile-time cost.

How to Interleave Elaboration and Typechecking. Most tailorings in this paper happen before typechecking, which leaves them free to accept DSL syntax but also forces them to accommodate ill-typed or even ill-formed programs. In Scala, typechecking happens before and after tailoring. Further research is needed to weigh the strengths of each approach on concrete examples. On a related note, several features in Typed Racket including `match` and list comprehensions are implemented as macros and therefore get typechecked only after expansion. The typechecker struggles to reason about this post-expansion code. A source-level type analysis may be more straightforward.

Toward a Proof API. Typechecking before expansion raises the question of how to share results with the host type system. The tailorings in this paper either produce simple code or use casts to share results. Other tailoring-adjacent systems, such as the units-of-measure Haskell plugin [43], merely assert results to the host language. One avenue for enhancement is to equip a standard typechecker with a tactic language (e.g., [37, 63]) to support proof objects. The System DE calculus implements a similar design for termination proofs [61]; it contains a sublanguage for constructing proof terms that the typechecker can accept and erase during compilation.

8 Discussion

Type tailoring is a lightweight way to grow [47] a type system by equipping it with additional expressiveness. Critically, tailorings are mere library code produced by and for ordinary users, rather than by compiler engineers. Additionally, tailorings cooperate with the existing type system and are composable with one another. Since tailorings run before the typechecker, host-language typechecking provides a basic correctness guarantee; for further assurance, this paper also presents a framework for reasoning about behavioral changes that tailorings may introduce. Finally, since a type tailoring leverages the host’s metaprogramming system, no extra effort is needed to integrate it into a language’s build system – unlike what a bespoke static analysis might require.

Although this paper has shown that support from a metaprogramming system empowers end users to build a type tailoring system, it would be interesting to see how first-class support for tailoring might improve its expressiveness and efficiency. No programming language includes type tailoring as an official, documented aspect of the language, but programmers clearly benefit from what support exists anyway. The many forms of tailoring in the programming language landscape give ample evidence that tailoring is a useful idea. The API blueprints presented here may allow these related efforts to build on one another.

In summary, this paper provides a research foundation that has takeaways for end users, language designers, and the authors of tailorings:

- For users, the examples in Section 2 show that type tailoring balances ease of use with some expressiveness of dependent types.
- For designers, the analysis in Section 3 explains why a powerful metaprogramming system is desirable. Tailoring in Julia, for example, falls short of what it could achieve with *cooperating API* features (Figure 1).
- For authors of tailorings, the guidelines in Section 5 separate well-reasoned tailorings from arbitrary reprogramming of the compiler front end.

Type tailoring leads to a broader perspective on what metaprogramming for types can achieve. It can facilitate maintainability and reliability for end users, help researchers prototype type system ideas, and reduce demands on the core type system.

References

- 1 Michael D. Adams. Towards the essence of hygiene. In *POPL*, pages 457–469. ACM, 2015. doi:[10.1145/2676726.2677013](https://doi.org/10.1145/2676726.2677013).
- 2 Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. Towards certified meta-programming with typed Template-Coq. In *ITP*, pages 20–39. Springer, 2018. doi:[10.1007/978-3-319-94821-8_2](https://doi.org/10.1007/978-3-319-94821-8_2).
- 3 Lennart Augustsson. Cayenne—a language with dependent types. In *ICFP*, pages 239–250. ACM, 1998. doi:[10.1145/289423.289451](https://doi.org/10.1145/289423.289451).

- 4 Christiaan Baaij. GHC type checker plugins: adding new type-level operations, 2016. . Accessed 2016-06-30. URL: <http://christiaanb.github.io/posts/type-checker-plugin/>.
- 5 Langston Barrett, David Thrane Christiansen, and Samuel Gélineau. Predictable macros for Hindley–Milner. In *TyDE*, 2020. Extended abstract.
- 6 Gary Bernhardt. Software: `static-path`. . Accessed 2024-01-17. URL: <https://github.com/garybernhardt/static-path>.
- 7 Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017. doi:10.1137/141000671.
- 8 Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *OOPSLA*, pages 215–230, 1993. doi:10.1145/165854.165893.
- 9 Eugene Burmako. Scala macros: Let our powers combine!: On how rich syntax and static types work with metaprogramming. In *SCALA*, pages 3:1–3:10. ACM, 2013. doi:10.1145/2489837.2489840.
- 10 Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5):509–543, 2009. doi:10.1017/S0956796809007205.
- 11 Stephen Chang, Michael Ballantyne, Milo Turner, and William J. Bowman. Dependent type systems as macros. *PACMPL*, 4(POPL):3:1–3:29, 2020. doi:10.1145/3371071.
- 12 Stephen Chang, Alex Knauth, and Ben Greenman. Type systems as macros. In *POPL*, pages 694–705. ACM, 2017. doi:10.1145/3009837.3009886.
- 13 David R. Christiansen and Edwin C. Brady. Elaborator reflection: Extending Idris in Idris. In *ICFP*, pages 284–297. ACM, 2016. doi:10.1145/2951913.2951932.
- 14 David Raymond Christiansen. Dependent type providers. In *WGP*, pages 25–34. ACM, 2013. doi:10.1145/2502488.2502495.
- 15 William D. Clinger and Jonathan Rees. Macros that work. In *POPL*, pages 155–162. ACM, 1991. doi:10.1145/99583.99607.
- 16 William D. Clinger and Mitchell Wand. Hygienic macro technology. *PACMPL*, 4(HOPL):80:1–80:110, 2020. doi:10.1145/3386330.
- 17 Clojure Contributors. Software: Clojure/tools.macro. . Accessed 2024-01-18. URL: <https://github.com/clojure/tools.macro>.
- 18 Clojure Contributors. Clojure macros, 2024. . Accessed 2024-01-17. URL: <https://clojure.org/reference/macros>.
- 19 Clojure Contributors. Clojure metadata documentation, 2024. . Accessed 2024-01-17. URL: <https://clojure.org/reference/metadata>.
- 20 Ryan Culpepper. Fortifying macros. *Journal of Functional Programming*, 22(4-5):439–476, 2012. doi:10.1017/S0956796812000275.
- 21 Ryan Culpepper, Sam Tobin-Hochstadt, and Matthew Flatt. Advanced macrology and the implementation of Typed Scheme. In *SFP. Université Laval, DIUL-RT-0701*, pages 1–14, 2007. URL: <http://www2.ift.ulaval.ca/~dadub100/sfp2007/procPaper1.pdf>.
- 22 Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kivanç Muşlu, and Todd W. Schiller. Building and using pluggable type checkers. In *ICSE*, pages 681–690, 2011. doi:10.1145/1985793.1985889.
- 23 R. Kent Dybvig. *Chez Scheme Users Guide*. Cadence Research Systems, 2nd edition, 2011.
- 24 Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A metaprogramming framework for formal verification. *PACMPL*, 1(ICFP):34:1–34:29, 2017. doi:10.1145/3110278.
- 25 ECMA International. ECMAScript language specification: 11.2.2 strict mode code, 2024. URL: <https://262.ecma-international.org/15.0/index.html#sec-strict-mode-code>.
- 26 Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. In *Haskell*, pages 117–130. ACM, 2012. doi:10.1145/2364506.2364522.
- 27 Elixir Contributors. Elixir macro documentation, 2024. . Accessed 2024-01-17. URL: <https://hexdocs.pm/elixir/1.16/Macro.html>.

- 28 Elixir Contributors. Elixir standard library, 2024. . Accessed 2024-01-17. URL: <https://hexdocs.pm/elixir/1.16.0/Kernel.html>.
- 29 Matthias Felleisen. Software: 7GUI, 2020. . Accessed 2023-12-21. URL: <https://github.com/mfelleisen/7GUI>.
- 30 David Fisher and Olin Shivers. Building language towers with Ziggurat. *Journal of Functional Programming*, 18(5-6):707–780, 2008. doi:[10.1017/S0956796808006928](https://doi.org/10.1017/S0956796808006928).
- 31 Matthew Flatt. Composable and compilable macros: You want it when? In *ICFP*, pages 72–83. ACM, 2002. doi:[10.1145/581478.581486](https://doi.org/10.1145/581478.581486).
- 32 Matthew Flatt. Binding as sets of scopes. In *POPL*, pages 705–717, 2016. doi:[10.1145/2837614.2837620](https://doi.org/10.1145/2837614.2837620).
- 33 Matthew Flatt, Taylor Allred, Nia Angle, Stephen De Gabrielle, Robert Bruce Findler, Jack Firth, Kiran Gopinathan, Ben Greenman, Siddhartha Kasivajhula, Alex Knauth, Jay McCarthy, Sam Phillips, Sorawee Porncharoenwase, Jens Axel Søgaard, and Sam Tobin-Hochstadt. Rhombus: A new spin on macros without all the parentheses. *PACMPL*, 7(OOPSLA2), 2023. doi:[10.1145/3622818](https://doi.org/10.1145/3622818).
- 34 Matthew Flatt, Ryan Culpepper, David Darais, and Robert Bruce Findler. Macros that work together: Compile-time bindings, partial expansion, and definition contexts. *Journal of Functional Programming*, 22(2):181–216, 2012. doi:[10.1017/S0956796812000093](https://doi.org/10.1017/S0956796812000093).
- 35 Kimball Germane, Jay McCarthy, Michael D. Adams, and Matthew Might. Demand control-flow analysis. In *VMCAI*, pages 226–246. Springer, 2019. doi:[10.1007/978-3-030-11245-5_11](https://doi.org/10.1007/978-3-030-11245-5_11).
- 36 GitHub. GitHub search: `use Phoenix.VerifiedRoutes` in Elixir, 2024. . Accessed 2024-01-17. URL: https://github.com/search?q=%22use+Phoenix.VerifiedRoutes%22+AND+%22def+verified_routes%22+language%3AElixir&type=code.
- 37 Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. How to make ad hoc proof automation less ad hoc. In *ICFP*, pages 163–175. ACM, 2011. doi:[10.1145/2034773.2034798](https://doi.org/10.1145/2034773.2034798).
- 38 Thomas R. G. Green and Marian Petre. Usability analysis of visual programming environments: A ‘cognitive dimensions’ framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996. doi:[10.1006/JVLC.1996.0009](https://doi.org/10.1006/JVLC.1996.0009).
- 39 Michael Greenberg. The dynamic practice and static theory of gradual typing. In *SNAPL*, pages 6:1–6:20. Schloss Dagstuhl, 2019. doi:[10.4230/LIPICS.SNAPL.2019.6](https://doi.org/10.4230/LIPICS.SNAPL.2019.6).
- 40 Ben Greenman. Type Tailoring, 2017. . Accessed 2024-01-08. URL: <https://blog.racket-lang.org/2017/04/type-tailoring.html>.
- 41 Ben Greenman. Trivial: Type tailored library functions, 2020. . Accessed 2024-01-07. URL: <https://docs.racket-lang.org/trivial/index.html>.
- 42 Ben Greenman. GTP benchmarks for gradual typing performance. In *REP*, pages 102–114. ACM, 2023. doi:[10.1145/3589806.3600034](https://doi.org/10.1145/3589806.3600034).
- 43 Adam Gundry. A typechecker plugin for units of measure: Domain-specific constraint solving in GHC Haskell. In *Haskell*, pages 11–22. ACM, 2015. doi:[10.1145/2804302.2804305](https://doi.org/10.1145/2804302.2804305).
- 44 Susumu Hayashi. Singleton, union and intersection types for program extraction. *Information and Computation*, 109(1/2):174–210, 1994. doi:[10.1006/INCO.1994.1016](https://doi.org/10.1006/INCO.1994.1016).
- 45 David Herman and Philippe Meunier. Improving the static analysis of embedded languages via partial evaluation. In *ICFP*, pages 16–27. ACM, 2004. doi:[10.1145/1016850.1016857](https://doi.org/10.1145/1016850.1016857).
- 46 Joel Jakubovic, Jonathan Edwards, and Tomas Petricek. Technical dimensions of programming systems. *Programming*, 7(3), 2023. doi:[10.22152/PROGRAMMING-JOURNAL.ORG/2023/7/13](https://doi.org/10.22152/PROGRAMMING-JOURNAL.ORG/2023/7/13).
- 47 Guy L. Steele Jr. Growing a language. In *Addendum to OOPSLA*. ACM, 1998. doi:[10.1145/346852.346922](https://doi.org/10.1145/346852.346922).
- 48 Julia Contributors. Julia AST documentation, 2024. . Accessed 2024-01-17. URL: <https://docs.julialang.org/en/v1/devdocs/ast/>.
- 49 Julia Contributors. Julia metaprogramming, 2024. . Accessed 2024-01-17. URL: <https://docs.julialang.org/en/v1/manual/metaprogramming/>.

- 50 Milod Kazerounian, Sankha Narayan Guria, Niki Vazou, Jeffrey S. Foster, and David Van Horn. Type-level computations for Ruby libraries. In *PLDI*, pages 966–979. ACM, 2019. doi: 10.1145/3314221.3314630.
- 51 Daniel Keep and Lukas Wirth. The little book of Rust macros, 2024. . Accessed 2024-01-17. URL: <https://veykril.github.io/tlborm/proc-macros/hygiene.html>.
- 52 Oleg Kiselyov. Reconciling abstraction with high performance: A MetaOCaml approach. *Foundations and Trends® in Programming Languages*, 5(1):1–101, 2018. doi:10.1561/2500000038.
- 53 Eugen Kiss. 7GUIs: A GUI programming benchmark. . Accessed 2023-12-21. URL: <https://eugenkiss.github.io/7guis/>.
- 54 Eugen Kiss. *Comparison of Object-Oriented and Functional Programming for GUI Development*. PhD thesis, Leibniz Universitaet Hannover, 2014.
- 55 Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce F. Duba. Hygienic macro expansion. In *LFP*, pages 151–161. ACM, 1986. doi:10.1145/319838.319859.
- 56 Eugene E. Kohlbecker and Mitchell Wand. Macro-by-example: Deriving syntactic transformations from their specifications. In *POPL*, pages 77–84. ACM, 1987. doi:10.1145/41625.41632.
- 57 Pepijn Kokke and Wouter Swierstra. Auto in Agda - programming proof search using reflection. In *MPC*, pages 276–301. Springer, 2015. doi:10.1007/978-3-319-19797-5_14.
- 58 Joomy Korkut and David Thrane Christiansen. Extensible type-directed editing. In *TyDe*, pages 38–50. ACM, 2018. doi:10.1145/3240719.3241791.
- 59 LaunchBadge. Software: SQLx, 2023. . Accessed 2024-01-17. URL: <https://github.com/launchbadge/sqlx>.
- 60 Fredrik Lindblad and Marcin Benke. A tool for automated theorem proving in Agda. In *TYPES*, pages 154–169. Springer, 2004. doi:10.1007/11617990_10.
- 61 Yiyun Liu and Stephanie Weirich. Dependently-typed programming with logical equality reflection. *PACMPL*, 7(ICFP):210:1–210:37, 2023. doi:10.1145/3607852.
- 62 Florian Lorenzen and Sebastian Erdweg. Sound type-dependent syntactic language extension. In *POPL*, pages 204–216. ACM, 2016. doi:10.1145/2837614.2837644.
- 63 Gregory Malecha and Jesper Bengtson. Extensible and efficient automation through reflective tactics. In *ESOP*, pages 532–559. Springer, 2016. doi:10.1007/978-3-662-49498-1_21.
- 64 Michał Marczyk. Answer to "does Clojure have identifier macros?". . Accessed 2024-01-18. URL: <https://stackoverflow.com/a/33426863/7327755>.
- 65 Chris McCord. Phoenix 1.7.0 released: Built-in Tailwind, Verified Routes, LiveView Streams, and what's next, 2023. . Accessed 2024-01-17. URL: <https://phoenixframework.org/blog/phoenix-1.7-final-released>.
- 66 Alp Mestanogullari, Sönke Hahn, Julian K. Arni, and Andres Löh. Type-level web APIs with Servant: An exercise in domain-specific generic programming. In *WGP*, pages 1–12. ACM, 2015. doi:10.1145/2808098.2808099.
- 67 Ana L. Milanova and Wei Huang. Inference and checking of context-sensitive pluggable types. In *FSE*, page 26. ACM, 2012. doi:10.1145/2393596.2393626.
- 68 David A. Moon. MACLISP reference manual, Revision 0. Technical report, MIT Project MAC, 1974.
- 69 Multiple Authors. Language: Macros, 2023. . Accessed 2024-01-17. URL: <https://clojure-doc.org/articles/language/macros/>.
- 70 OCaml Contributors. OCaml PPX, 2024. . Accessed 2024-01-17. URL: <https://ocaml.org/docs/metaprogramming>.
- 71 Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Safely composable type-specific languages. In *ECOOP*, pages 105–130. Springer, 2014. doi:10.1007/978-3-662-44202-9_5.
- 72 Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *ISSTA*, pages 201–212. ACM, 2008. doi: 10.1145/1390630.1390656.

- 73 Lionel Parreaux. Squid—type-safe metaprogramming for Scala, 2024. . Accessed 2024-01-17. URL: <https://epfldata.github.io/squid/home.html>.
- 74 Lionel Parreaux, Antoine Voizard, Amir Shaikhha, and Christoph E. Koch. Unifying analytic and statically-typed quasiquotes. *PACMPL*, 2(POPL):13:1–13:33, 2018. doi:10.1145/3158101.
- 75 Tomas Petricek, Gustavo Guerra, and Don Syme. Types from data: Making structured data first-class citizens in F#. In *PLDI*, pages 477–490. ACM, 2016. doi:10.1145/2908080.2908115.
- 76 Phoenix Contributors. Phoenix verified routes, 2023. <https://hexdocs.pm/phoenix/Phoenix.VerifiedRoutes.html>. Accessed 2023-11-28.
- 77 Justin Pombrio and Shriram Krishnamurthi. Hygienic resugaring of compositional desugaring. In *ICFP*, pages 75–87. ACM, 2015. doi:10.1145/2784731.2784755.
- 78 Racket Contributors. Racket syntax properties documentation, 2024. . Accessed 2024-01-17. URL: <https://docs.racket-lang.org/reference/stxprops.html>.
- 79 Racket Contributors. Syntax transformers, 2024. . Accessed 2024-01-17. URL: <https://docs.racket-lang.org/reference/stxtrans.html>.
- 80 Rhombus Contributors. Rhombus documentation: Static and dynamic lookup, 2024. . Accessed 2024-01-17. URL: https://docs.racket-lang.org/rhombus/Static_and_Dynamic_Lookup.html.
- 81 Rhombus Contributors. Rhombus syntax object documentation, 2024. . Accessed 2024-01-17. URL: <https://docs.racket-lang.org/rhombus/stxobj.html>.
- 82 Tiark Rompf. Reflections on LMS: exploring front-end alternatives. In *SCALA*, pages 41–50. ACM, 2016. doi:10.1145/2998392.2998399.
- 83 Tiark Rompf and Nada Amin. A SQL to C compiler in 500 lines of code. *Journal of Functional Programming*, 29:e9, 2019. doi:10.1017/S0956796819000054.
- 84 Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In *GPCE*, pages 127–136. ACM, 2010. doi:10.1145/1868294.1868314.
- 85 Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In *POPL*, pages 497–510. ACM, 2013. doi:10.1145/2429069.2429128.
- 86 Rust Contributors. Rust macros, 2024. . Accessed 2024-01-17. URL: <https://doc.rust-lang.org/reference/procedural-macros.html>.
- 87 Scala 3 Contributors. Scala 3 macros, 2024. . Accessed 2024-01-17. URL: <https://docs.scala-lang.org/scala3/guides/macros/macros.html>.
- 88 Scala 3 Contributors. Scala 3 reference: Macros, 2024. . Accessed 2024-01-17. URL: <https://docs.scala-lang.org/scala3/reference/metaprogramming/macros.html>.
- 89 SciML Contributors. Software: OrdinaryDiffEq.jl, 2024. . Accessed 2024-01-17. URL: <https://github.com/SciML/OrdinaryDiffEq.jl>.
- 90 Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Haskell*, pages 1–16. ACM, 2002. doi:10.1145/581690.581691.
- 91 Olin Shivers. Control-flow analysis in Scheme. In *PLDI*, pages 164–174. ACM, 1988. doi:10.1145/53990.54007.
- 92 Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *SFP. University of Chicago, TR-2006-06*, pages 81–92, 2006. URL: <http://scheme2006.cs.uchicago.edu/scheme2006.pdf>.
- 93 Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *SNAPL*, pages 274–293. Schloss Dagstuhl, 2015. doi:10.4230/LIPICS.SNAPL.2015.274.
- 94 Suzanne Soy. Software: `type-expander`, 2020. . Accessed 2023-12-21. URL: <https://github.com/SuzanneSoy/type-expander>.

- 95 Eric Spishak, Werner Dietl, and Michael D. Ernst. A type system for regular expressions. In *FTfJP*, pages 20–26. ACM, 2012. doi:10.1145/2318202.2318207.
- 96 Antonis Stampoulis and Zhong Shao. VeriML: Typed computation of logical terms inside a language with effects. In *ICFP*, pages 333–344. ACM, 2010. doi:10.1145/1863543.1863591.
- 97 StaticArrays Contributors. StaticArrays, 2024. . Accessed 2024-01-17. URL: <https://juliahub.com/ui/Packages/General/StaticArrays/>.
- 98 Guy L. Steele, Jr. *Common Lisp*. Digital Press, 2nd edition, 1990.
- 99 T. Stephen Strickland, Sam Tobin-Hochstadt, and Matthias Felleisen. Practical variable-arity polymorphism. In *ESOP*, pages 32–46, 2009. doi:10.1007/978-3-642-00590-9_3.
- 100 Satish Thatte. Quasi-static typing. In *POPL*, pages 367–381, 1990. doi:10.1145/96709.96747.
- 101 Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *DLS*, pages 964–974, 2006. doi:10.1145/1176617.1176755.
- 102 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *POPL*, pages 395–406, 2008. doi:10.1145/1328438.1328486.
- 103 Sam Tobin-Hochstadt, Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa. Migratory typing: Ten years later. In *SNAPL*, pages 17:1–17:17. Schloss Dagstuhl, 2017. doi:10.4230/LIPICS.SNAPL.2017.17.
- 104 Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *PLDI*, pages 132–141, 2011. doi:10.1145/1993498.1993514.
- 105 José Valim. Nx: Numerical Elixir, 2023. . Accessed 2024-01-16. URL: <https://github.com/elixir-nx/nx>.
- 106 David Van Horn. Software: zombie, 2020. . Accessed 2023-02-20. URL: <https://github.com/philnguyen/soft-contract/tree/master/soft-contract/benchmark-contract-overhead>.
- 107 Stephanie Weirich. The influence of dependent types (keynote). *SIGPLAN Notices*, 52(1), 2017. doi:10.1145/3093333.3009923.
- 108 Ashton Wiersdorf, Stephen Chang, Matthias Felleisen, and Ben Greenman. Artifact for Type Tailoring (ECOOP 2024), July 2024. doi:10.5281/zenodo.12726060.
- 109 Hongwei Xi. Dependent ML: An approach to practical programming with dependent types. *Journal of Functional Programming*, 17(2):215–286, 2007. doi:10.1017/S0956796806006216.
- 110 Ningning Xie, Leo White, Olivier Nicole, and Jeremy Yallop. MacoCaml: Staging composable and compilable macros. *PACMPL*, 7(ICFP), 2023. doi:10.1145/3607851.

Higher-Order Specifications for Deductive Synthesis of Programs with Pointers

David Young* 

University of Kansas, Lawrence, KS, USA

Ziyi Yang* 

National University of Singapore, Singapore

Ilya Sergey 

National University of Singapore, Singapore

Alex Potanin 

Australian National University, Canberra, Australia

Abstract

Synthetic Separation Logic (SSL) is a formalism that powers SuSLik, the state-of-the-art approach for the deductive synthesis of provably-correct programs in C-like languages that manipulate heap-based linked data structures. Despite its expressivity, SSL suffers from two shortcomings that hinder its utility. First, its main specification component, inductive predicates, only admits *first-order* definitions of data structure shapes, which leads to the proliferation of “boiler-plate” predicates for specifying common patterns. Second, SSL requires *concrete* definitions of data structures to synthesise programs that manipulate them, which results in the need to change a specification for a synthesis task every time changes are introduced into the layout of the involved structures.

We propose to significantly lift the level of abstraction used in writing Separation Logic specifications for synthesis – both simplifying the approach and making the specifications more usable and easy to read and follow. We avoid the need to repetitively re-state low-level representation details throughout the specifications – allowing the reuse of different implementations of the same data structure by abstracting away the details of a specific layout used in memory. Our novel *high-level front-end language* called Pika significantly improves the expressiveness of SuSLik.

We implemented a layout-agnostic synthesiser from Pika to SuSLik enabling push-button synthesis of C programs with in-place memory updates, along with the accompanying full proofs that they meet Separation Logic-style specifications, from high-level specifications that resemble ordinary functional programs. Our experiments show that our tool can produce C code that is comparable in its performance characteristics and is sometimes faster than Haskell.

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

Keywords and phrases Program Synthesis, Separation Logic, Functional Programming

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.45

Related Version *Extended Version:* <https://arxiv.org/abs/2407.09143> [14]

Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact):* <https://doi.org/10.4230/DARTS.10.2.25>

Software (Source Code): <https://github.com/roboguy13/PikaC>

archived at sw.h:1/dir:b11b817c991d60c0916c4a3341a9932a4c4080ed

Software (Docker for Artifact evaluation): <https://zenodo.org/records/10558356>

Funding This work was partially supported by a Singapore Ministry of Education (MoE) Tier 3 grant “Automated Program Repair” MOE-MOET32021-0001 and MoE Tier 1 grant T1 251RES2108 “Automated Proof Evolution for Verified Software Systems”.

* Contributed equally to this work.

 © David Young, Ziyi Yang, Ilya Sergey, and Alex Potanin;
licensed under Creative Commons License CC-BY 4.0
38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 45; pp. 45:1–45:26



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



Acknowledgements We thank the anonymous ECOOP 2024 PC and AEC reviewers for their constructive and insightful comments.

1 Introduction

Recent advances in program synthesis have allowed programmers to concentrate on stating precise specifications – leaving the job of generating provably correct and efficient imperative code to the synthesiser, such as SuSLik [6, 9, 13]. Such specifications are usually expressed using (Synthetic) Separation Logic [7, 10] that while hugely successful in verifying properties of pointer-manipulating programs remains out of reach to many mainstream developers. As the programs grow in complexity, such SSL specifications can become exceedingly verbose and complex – making the job of specification writer especially error-prone and defeating the purpose of a *usable proof automation toolchain*.

The power of Separation Logic (SL) specifications for the tasks of both verification and synthesis, is its mechanism of *inductive predicates* that concisely capture the shape of possibly recursive pointer-based tree-like data structures, determining both induction schemes for verification and the shape of recursion for the synthesis tasks of the programs that manipulate such data structures [10]. The surprising ability of SL specifications to capture precisely the logic of a desired program to be synthesised in the logical assertions comes at the price of the involved inductive predicates being (a) *first-order* and (b) somewhat *low-level*, with both these aspects posing limitations to the usability of SL-based program synthesis.

The first-order nature of the predicates means, for instance, that synthesis tasks that involve several data structures with very similar heap layouts would require to use *different* predicate definitions. As a specific example, consider a task synthesising two functions, *f* and *g*. Both *f* and *g* take as an argument a pointer to a linked list of integers; *f* increments all elements of the list by one, while *g* multiplies all its elements by two. To specify these two tasks, the state-of-the-art tools for program synthesis based on SL specifications require the user to provide, in addition to the pre-/postconditions, *three* different inductive predicates: one for an arbitrary list, another for a list that carries a known payload, with each element incremented by one, and the final capturing the multiplication of each element by two.

The second aspect, *i.e.*, the low-level nature of the SL inductive predicates used for synthesis, shows up when we try to *rewrite* an already specified synthesis task for a data structure with a slightly different layout. As an example, imagine defining the task of concatenating two lists. It is natural to expect that the specification will look very similar for both singly and doubly linked lists. Yet, since those are two different structures, with two different layouts, the user would require to supply two different task specifications.

A seasoned programmer would immediately notice that both issues (a) and (b) very much resemble the struggle that one faces when programming in a language that does not provide certain abstractions, which are nowadays mostly taken for granted: *higher-order functions* and *abstract data types*. Our first example could be streamlined should the SL-based specification language offer a way to define a function similar to *List.map*, available in all popular functional programming languages, so it could be used to concisely express the two scenarios of manipulating with the payload of a list’s elements. The second example would benefit from the ability to specify concatenation of *abstract* lists, while separately handling manipulations with single- or doubly-linked lists in terms of their memory *layouts*.



■ **Figure 1** Pika translation pipeline.

Key Ideas

The two challenges faced by the SSL specification language for the synthesis of heap-manipulating programs – the need for higher-order functions and abstract data types – have provided the primary motivation for this work.

As a solution, we developed **Pika**: a high-level front-end language and specification translation framework built on top of the state-of-the-art SL-based program synthesiser **SuSLik**.¹ **SuSLik** is based on a variant of separation logic called *synthetic separation logic* or **SSL** [9]. **Pika** has a syntax similar to popular functional programming languages and features specification-level higher-order functions on Algebraic Data Types (ADTs). In addition to being more succinct in comparison to Synthetic Separation Logic (SSL), the specification formalism of **SuSLik**, **Pika** also addresses its reusability issues outlined above. First, the use of specification-level higher-order functions allows the user to abstract over the specific properties of the payloads of the heap-based data structures, thus, generalising existing inductive predicates so they could be employed in a wider range of synthesis tasks. Second, by manipulating ADTs, the synthesis specifications do not need to deal with the specific memory layouts of the data structures. This separates the specification of the tasks that operate on those data structures from the low-level details of their memory representations.

The **Pika** pipeline is depicted in Figure 1. As our goal was to extend the expressivity of **SuSLik** by giving it a high-level specification language while retaining its meta-theoretical guarantees (*i.e.*, the certifiable correctness of the programs it synthesises), we had to overcome several technical obstacles when designing **Pika** and implementing it on top of **SuSLik**. First, we had to formally define the operational semantics of **Pika** and its translation to SSL-based specifications of **SuSLik**, defining the corresponding soundness result, relating the behaviour of programs eventually synthesised to that of their high-level counterparts (Section 3). Second, to support the higher-order specifications of **Pika**, we introduced conservative extensions to **SuSLik**'s specification language as well as to its deductive synthesis rules, to make it capable of handling pre-/postconditions with first-class functions operating on the payload of heap-stored data types (Section 4).

Pika as a Programming Language

Given the close similarity of **Pika**, our new specification language, to general-purpose functional programming languages, such as Haskell, it is natural to wonder whether it's possible to leverage its underlying synthesis pipeline as a way to produce efficient imperative programs in a language such as C from equivalent high-level functional programs. In other words, if we decide to use a combination **Pika** + **SuSLik** as a *compiler*, would it be a viable replacement to many-decades old tools such as Glasgow Haskell Compiler (GHC) [3], for producing efficient runnable code? To answer this question, we have conducted evaluation on several list- and tree-manipulating benchmarks, comparing the performance of verified C code, emitted by **SuSLik** from **Pika** specifications, to that of executables produced for equivalent tasks by GHC.

¹ Both susliks and pikas are Central Asian mammals. Pikas might look similar to susliks, but are more nimble and have longer life expectancy.

Our preliminary results are encouraging: thanks to avoiding unnecessary allocation and using destructive heap updates whenever possible, the C code synthesised by Pika + SuSLik outperforms the GHC output (with compiler optimisations flags turned) in the majority of list-manipulating benchmarks we've tried; our synthesis tool also produces strictly more performant C code for tree-manipulating benchmarks when compared to the corresponding Haskell programs, compiled by GHC without or with optimisations (Section 5). In particular, we specifically observe this when we compare C code synthesised by Pika + SuSLik with no C compiler optimisation flags to GHC compiled code with no GHC compiler flags.

Contributions

In this work, we make the following contributions:

- We address the expressivity limitations of SSL, the specification formalism of the state-of-the-art deductive synthesis tool SuSLik by developing Pika— a high-level specification language with higher-order functions and abstract data types.
- We formally define the operational semantics of Pika and prove the soundness of translation from Pika to pre-/postconditions in SSL.
- We develop an extension to SuSLik's specification and synthesis mechanism that enables translation from Pika specifications featuring first-class functions.
- We observe that the synthesis tool resulting from the combination Pika + SuSLik enables certified memory-layout-agnostic compilation from a functional specification to C code.
- We report on the evaluation of the Pika regarding its expressiveness and performance. In particular, we show that the C code it produces frequently outperforms equivalent Haskell programs compiled by GHC.

2 Overview

2.1 Background

The Pika language is translated into SuSLik [9], which is a program synthesis tool that uses Synthetic Separation Logic (SSL)-a variant of Hoare-style [4] Separation Logic (SL) [7].

A synthesis task specification in SuSLik is given as a function signature together with a pair of pre- and post-conditions, which are both SL assertions [9]. The synthesiser generates code that satisfies the given specification, along with the SL *proof* of its correctness by searching in a space of proofs that can be derived by using the rules of the underlying logic [13]. A distinguishing feature of *Synthetic Separation Logic* is the format of its assertions. An SSL assertion consists of two parts: a *pure part* and a *spatial part*. The pure part is a Boolean expression constraining the variables of the specification using a few basic relations, like equality and the less-than relation for integers. The spatial part is a *symbolic heap*, which consists of a list of *heaplets* separated by the * symbol.

Each heaplet takes on one of the following forms [9]:

- **emp**: This represents the empty heap. It is also the left and right identity for *.
- $\ell \mapsto a$: This asserts that memory location ℓ points to the value a . It also asserts that the location ℓ is accessible.
- $[\ell, \iota]$: At memory location ℓ , there is a block of size ι .
- $p(\bar{\phi})$: This is an application of the **inductive predicate** p to the arguments $\bar{\phi}$. An inductive predicate has a collection of branches, guarded by Boolean expressions (conditions) on the parameters. The body of each branch is an SSL assertion. The assertion associated with the first branch condition that is satisfied is used in place of the application. Note that inductive predicates can be (and often are) recursively defined.

Variable	x, y Alpha-numeric identifiers $\in \text{Var}$
Size, offset	n, ι Non-negative integers
Expression	$e ::= 0 \mid \text{true} \mid x \mid e = e \mid e \wedge e \mid \neg e \mid d$
\mathcal{T} -expr.	$d ::= n \mid x \mid d + d \mid n \cdot d \mid \{\} \mid d \mid \dots$
Command	$c ::= \text{let } x = *(\mathbf{x} + \iota) \mid *(\mathbf{x} + \iota) = e \mid$ $\quad \quad \quad \text{let } x = \text{malloc}(n) \mid \text{free}(x) \mid \text{error} \mid f(\overline{e_i})$
Program	$\Pi ::= \overline{f(\overline{x_i}) \{ c \}} ; c$
Logical variable	ν, ω
Cardinality variable	α
\mathcal{T} -term	$\kappa ::= \nu \mid e \mid \dots$
Pure logic term	$\phi, \psi, \chi ::= \kappa \mid \phi = \phi \mid \phi \wedge \phi \mid \neg \phi$
Symbolic heap	$P, Q, R ::= \text{emp} \mid \langle e, \iota \rangle \mapsto e \mid [e, \iota] \mid p^\alpha(\overline{\phi_i}) \mid P * Q$
Heap predicate	$\mathcal{D} ::= p^\alpha(\overline{x_i}) : e_j \Rightarrow \exists \overline{y}. \{ \chi_j; R_j \}$
Assertion	$\mathcal{P}, \mathcal{Q} ::= \{ \phi; P \}$
Environment	$\Gamma ::= \forall \overline{x_i}. \exists \overline{y_j}.$
Context	$\Sigma ::= \overline{\mathcal{D}}$
Synthesis goal	$\mathcal{G} ::= P \rightsquigarrow Q$

Figure 2 Syntax of Synthetic Separation Logic.

The general form of an SSL assertion is $(p; h_1 * h_2 * \dots * h_n)$, where p is the pure part and h_1, h_2, \dots, h_n are the heaplets which are the conjuncts that make up a separated conjunction of the spatial part. $*$ is *separating conjunction*: $h_1 * h_2$ means that the heaplets h_1 and h_2 apply to disjoint parts of the heap. A syntax definition for SSL is given in Figure 2, which is adapted from *Cyclic Program Synthesis* by Itzhaky *et al.* [6]. We will also use the symbol $**$ for separating conjunction and the symbol \rightarrowtail for \mapsto .

As the specification language of SuSLik, SSL serves as the compilation target for the Pika language. From there, executable programs are generated through SuSLik’s program synthesis. Consider a program that takes an integer x and a result in location r and stores $x + 1$ at location r . This can be written as the SuSLik specification:

```
void add1Proc(int x, loc r)
{ r :> 0 }
{ y == x + 1 ; r :> y }
{ ?? }
```

This example can be written as follows in our tool:

```
add1Proc : Int -> Int;
add1Proc x := x + 1;
```

In contrast with SuSLik spec, the Pika one requires no direct manipulation with pointers.

2.2 The Pika Language

While SSL provides a specification language that allows tools like SuSLik to synthesise code, it is only able to express specifications as pointers. This is useful for some applications, such as embedded systems, but it does not provide any high-level abstractions. As a result, every part of a specification is tailored to a specific memory representation of each data structure involved. To address this shortcoming, we introduce a language with algebraic data types that gets translated into SSL specifications. Additionally, we introduce a language construct that allows the programmer to specify a *memory representation* of an algebraic

```

data List := Nil | Cons Int List;

S11 : List >-> layout[x];
S11 (Nil) := emp;
S11 (Cons head tail) := x :-> head, (x+1) :-> tail, S11 tail;

```

Figure 3 List algebraic data type together with its singly-linked list layout S11.

data type called a *layout*. The distinction between algebraic data types and layouts provides the separation of concerns between the low-level representation of a data structure and code that manipulates it at a high level.

Syntactically, Pika resembles a functional programming language of the Miranda [12] and Haskell [5] lineage. It supports algebraic data types, pattern matching at top-level function definitions (though not inside expressions) and Boolean guards. The primary difference arises due to the existence of layouts and the fact that the language is compiled to an SSL specification rather than executable code. Beyond algebraic data types and layouts, Pika has a built-in type for integers as well as Booleans.

Functions in Pika are only defined by their operations on algebraic data types. Thus, all function definitions are “layout-polymorphic” over the particular choices of layouts for their arguments and result. Giving a layout polymorphic function, a particular choice of layouts is called “instantiation”. Specifying the layout of a non-function value is called “lowering.”

The code generator is instructed to generate a SuSLik specification for a certain function at a certain instantiation by using a `%generate` directive. For example, if there is a function definition with the type signature `mapAdd1 : List -> List`, a line reading `%generate mapAdd1 [S11] S11` would instruct the Pika compiler to generate the SuSLik inductive predicate corresponding to `mapAdd1` instantiated to the `S11` layout for both its argument and its result. An example of an ADT definition and a corresponding layout definition is given in Figure 3. There is one unusual part of the syntax in particular that requires further explanation: layout type signatures. A layout definition consists of a *layout type signature* and a pattern match (much like a function definition), with lists of SSL heaplets on the right-hand sides. A layout type signature has a special form `A : α → layout[x]`. This says that the layout `A` is for the algebraic data type `α` and the SSL output variable `x` denoting the “head” pointer of the respective structure.

2.3 Pika by Example

We demonstrate the characteristic usages of Pika by a series of examples. In these examples, we will often make use of the `List` algebraic data type and its `S11` layout from Figure 3. A simple example of Pika code that illustrates algebraic data types and layouts is a function which creates a singleton list out of the given integer argument:

```

%generate singleton [Int] S11

singleton : Int -> List;
singleton x := Cons x (Nil);

```

This gets compiled to the following SuSLik specification (modulo auto-generated names):

```

predicate singleton(int p, loc r) {
| true => { r :-> p ** (r+1) :-> 0 ** [r,2] }
}

```

```

predicate sll(loc x) {
| x == 0 => { emp }
| not (x == 0) => { [x, 2] ** x :-> v ** (x+1) :-> nxt ** sll(nxt) }
}

predicate mapAdd1(loc x, loc r) {
| x == 0 => { emp }
| not (x == 0) => { [x, 2] ** x :-> v ** (x+1) :-> xNxt ** [r, 2]
                      ** r :-> (v+1) ** (r+1) :-> rNxt ** mapAdd1(xNxt, rNxt) }
}

void mapAdd1_fn(loc x, loc y)
{ sll(x) ** y :-> 0 }
{ y :-> r ** mapAdd1(x, r) }
{ ?? }

```

Figure 4 Specifying a function that adds one to each element of a singly-linked list in SuSLik.

A slightly more complicated example comes from trying to write a functional-style `map` function directly in SuSLik. Consider a function which adds 1 to each integer in a list of integers. Considering the list implementation to be a singly-linked list with a fixed layout, one way to express this in SuSLik is shown in Figure 4. In the `mapAdd1` predicate, the input list is given as the `x` parameter and the `r` parameter points to the output list. In the non-null case, the head of `r` is required to be the successor of the head of `x`. The predicate is then applied recursively to the tails.

Note that inductive predicates are used for *two* different purposes: the `sll` inductive predicate describes a singly-linked list data structure, while the `mapAdd1` inductive predicate describes how the input list relates to the output list. Both are used in the specification of `mapAdd1_fn`: `sll` in the precondition and `mapAdd1` in the postcondition.

Using the `mapAdd1` inductive predicate gives us two advantages over attempting to put the SSL propositions directly into the postcondition of `mapAdd1_fn`:

1. We are able to express a conditional on the shape of the list. This is much like pattern matching in a language with algebraic data types, but we are examining the pointer involved directly.
2. We are able to express *recursion* part of the postcondition: the `mapAdd1` inductive predicate refers to itself in the `not (x == 0)` branch.

These two features are both reminiscent of features common in functional programming languages: pattern matching and recursion. However, there are still some significant differences:

- In traditional pattern matching, the underlying memory representation of the data structure is not exposed.
- Compared to a functional programming language, the meaning of the specification is more obscured. It is necessary to think about the structure of the linked data structure to determine what the specification is saying. This is related to the first point: The memory representation is front-and-center.
- In many functional languages, mutation is either restricted or generally discouraged. In SuSLik, mutation is commonplace.

Suppose we want to write the functional program that corresponds to this specification. One way to do this in a Haskell-like language is by using the `List` type from Figure 3.

```

mapAdd1_fn : List -> List;
mapAdd1_fn (Nil) := 0;
mapAdd1_fn (Cons head tail) := Cons (head + 1) (mapAdd1_fn tail);

```

The only missing information is the memory representation of the `List` data structure. We do not want the `mapAdd1_fn` implementation to deal with this directly, however. We want to separate the more abstract notions of pattern matching and constructors from the concrete memory layout that the data structure has.

To accomplish this, we now extend the code with the definition of `S11` from Figure 3. `S11` is a *layout* for the algebraic data type `List`. Now we have all of the information of the original specification but rearranged so that the low-level memory layout is separated from the rest of the code. This separation brings us to an important observation about the language, manifested throughout these examples: none of the function definitions need to *directly* perform any pointer manipulations. This is relegated entirely to the reusable layout definitions for the ADTs. The examples are written entirely as recursive functions that pattern match on, and construct, ADTs.

All that is left is to connect these two parts: the layouts and the function definitions. We instruct a `SuSLik` specification generator to generate a `SuSLik` specification from the `mapAdd1_fn` function using the `S11` layout:

```
%generate mapAdd1_fn [S11] S11
```

The `[S11]` part of the directive tells the generator which layouts are used for the arguments. In this case, the function only has one argument and the `S11` layout is used. The `S11` at the end specifies the layout for the result.

2.3.1 Synthesising the in-place map function

We can generalise our `mapAdd1` to map arbitrary `Int` functions over a list and then redefine `mapAdd1` using the new `map`.

```
%generate mapAdd1 [S11] S11

data List := Nil | Cons Int List;

S11 : List >-> layout[x];
S11 (Nil) := emp;
S11 (Cons head tail) := x :-> head, (x+1) :-> tail, S11 tail;

map : (Int -> Int) -> List -> List;
map f (Nil) := Nil;
map f (Cons x xs) := Cons (instantiate [Int] Int f x) (map f xs);

add1 : Int -> Int;
add1 x := x + 1;

mapAdd1 : List -> List;
mapAdd1 xs := instantiate [Int -> Int, S11] S11 map add1 xs;
```

The keyword `instantiate` gives specific layouts to use for the types in function applications, *e.g.*, if a function `g` has type `A -> B -> C -> D`, then `instantiate [L1, L2, L3] L4 g x y z` will use layout `L1` for type `A`, `L2` for type `B`, `L3` for type `C` and, finally, `L4` for the result type `D`, while applying the function `g` to the three arguments `x`, `y` and `z`.

This example makes use of `instantiate` in two places. The first one in the call of `instantiate [Int] Int f x`: the builtin `Int` layout is used for both the input and output. In this special case, the `Int` layout shares a name with the `Int` type that it represents. This is necessary since `instantiate` is used for all non-recursive (non constructor) calls.

```

predicate filterLt9(loc x, loc r) {
| (x == 0) => { r == 0 ; emp }
| not (x == 0) && head < 9 =>
  { x :-> head ** (x+1) :-> tail ** [x,2] ** filterLt9(tail, r) }
| not (x == 0) && not (head < 9) =>
  { x :-> head ** (x+1) :-> tail ** [x,2] ** filterLt9(tail, y)
    ** r :-> head ** (r+1) :-> y ** [r,2] } }

void filterLt9(loc x1, loc r)
{ S11(x1) ** r :-> 0 }
{ filterLt9(x1, r0) ** r :-> r0 }
{ ?? }

```

Figure 5 SuSLik specification of `filterLt9`, excluding `S11`, which is given in Figure 3.

In the second use, `instantiate [Int -> Int, S11] S11 map add1 xs`, we specify that the second argument uses the `S11` layout for the `List` type from Figure 3. We also give `S11` as the layout for the result of the call. Note that it is not necessary to use `instantiate` for the recursive call to `map`. This is because the appropriate layout is inferred for recursive calls.

The type signature of `mapAdd1` implies that it is layout polymorphic, as the type does not refer to any specific layout. It might be surprising that `instantiate` is required in the body of `mapAdd1` since the type signature of `mapAdd1` suggests that it is layout polymorphic and yet we must pick a specific `List` layout when we use `instantiate` to call `map`. This is because, in general, a call inside the body of some function `fn` might use any layout, even layouts that have no relation to the layouts that `fn` is instantiated to. Finally, please note that our benchmarks shown in Figure 1 include a more general version of `mapAdd`.

2.3.2 Guards

While we have a pattern-matching construct at the top level of a function definition, we have not seen a way to branch on a Boolean value so far. This is a feature that is readily available at the level of `SuSLik`, since the same conditional construct we use to implement pattern matching can also use other Boolean expressions.

We can expose this in the functional language using a *guard*, much like Haskell's guards. Suppose we want to write a specialised filter-like function. Specifically, we want a function that filters out all elements of a list that are less than 9. This is a specific example where the `SuSLik` specification is noticeably more difficult to read. For a `SuSLik` specification of this example, see Figure 5. On the other hand, an implementation of this in `Pika` is:

```

%generate filterLt9 [S11] S11

filterLt9 : List -> List;
filterLt9 (Nil) := Nil;
filterLt9 (Cons head tail)
| head < 9      := filterLt9 tail;
| not (head < 9) := Cons head (filterLt9 tail);

```

When translating a guarded function body, the translator takes the conjunction of the Boolean guard condition with the condition for the pattern match. Finally, please note that our benchmarks shown in Figure 1 include a more general version of `filterLt`.

While the SuSLik version of `filterLt9` requires working with pointers directly, the Pika version uses pattern matching and constructor application. This allows the Pika code to work independent of the layout used.

2.3.3 if-then-else

Another feature that is common in functional languages is `if-then-else` expressions. This has a straightforward translation into SuSLik. The `if-then-else` construct corresponds to SuSLik's C-like ternary operator. We can use this feature to implement the `even` function which produces 1 if the argument is even and 0 otherwise.

```
%generate even [Int] Int

even : Int -> Int;
even (n) := if (n % 2) == 0 then 1 else 0;
```

2.3.4 Using multiple layouts

To show the interaction between multiple algebraic data types, we write a function that follows the left branches of a binary tree and collects the values stored in those nodes into a list. This example demonstrates a binary tree algebraic data type and a layout that corresponds to it.

```
%generate leftList [TreeLayout] S11

data Tree := Leaf | Node Int Tree Tree;

TreeLayout : Tree -> layout[x];
TreeLayout (Leaf) := emp;
TreeLayout (Node payload left right) := x :-> payload, (x+1) :-> left,
(x+2) :-> right, TreeLayout left, TreeLayout right;

leftList : Tree -> List;
leftList (Leaf) := Nil;
leftList (Node a b c) := Cons a (leftList b);
```

2.3.5 Synthesising fold

A fold is a common kind of operation on a data structure in functional programming, where a binary function is applied to the elements of a data structure to create a summary value. For example, if the binary function is the addition function, it will give the sum of all the elements of the data structure. The classic example of such a fold is a fold on a list. In this example, we will write a right fold over a `List`.

```
%generate fold_List [Int, S11] Int
fold_List : Int -> List -> Int;
fold_List z (Nil) := z;
fold_List z (Cons x xs) :=
  instantiate [Int, Int] Int f x (fold_List z xs);
```

We will specifically look at the specialization where we use the addition function for `f` so that we can focus on way that layouts are used in the translation. This sort of specialization corresponds to defunctionalization. The compiler produces the following SuSLik specification for `fold_List`:

```
predicate fold_List(int i1, loc x, int i2) {
| x == 0 => { i2 == i1 ; emp }
| not (x == 0) => { (zz4 == i1) && ((zz5 == nxt13) &&
| (i2 == (h + b3))) ; [x,2] ** x :-> h ** (x + 1) :-> nxt13 **
| fold_List(zz4, zz5, b3) }
```

The first two parameters of the SuSLik predicate correspond to the two arguments of the Pika function. The final parameter of the predicate, `i2`, corresponds to the output of the Pika function. There are two cases:

1. The `x == 0` case corresponds to the `Nil` case in Pika. In this case, the pure part of the assertion (to the left of the semicolon) requires that the output is equal to the first parameter. This is because the Pika function returns the first parameter in its `Nil` case.
2. The `not (x == 0)` case corresponds to the `Cons` case. First, let's look at the spatial part (this is everything to the *right* of the semicolon). The pattern match destructures the `Cons` into its head and tail. Likewise, in the spatial part of the SuSLik predicate, we require that `x` points to `h` (the head) and `x + 1` points to `nxt13` (the tail). We also recursively call the predicate on the tail. The pure part does two things: it introduces new names for things (these are used internally) and it requires that the output `i2` is the sum of the head (`h`) and the value obtained from the recursive call (`b3`).

3 Formal Semantics of Pika

In this section, we have the following plan:

- We define abstract machine semantics for executing a subset of Pika programs. This semantics is given by the big-step relation \rightarrow which we define later.
- We define the translation from that subset of Pika into SSL. This translation is given by the function $\mathcal{T}[e]_{V,r}$ from Pika expressions into SSL propositions. The r is a variable name to be used in the resulting proposition and V is a collection of fresh names.
- We prove a soundness theorem. Given any well-typed expression e and an abstract machine reduction producing the store-heap pair (σ', h') , the SSL translation of e should be satisfied by SSL model (σ', h') . This is stated formally, and proven, in Theorem 3.

This subset of Pika does not have guards or conditional expressions, but it does have pattern matching. It also has the requirement that functions can only have one argument. Unlike the implementation, there is no elaboration. As a result, every algebraic data type value must be lowered to a specific layout at every usage and every function application must be explicitly instantiated with a layout for the argument and a layout for the result. We also limit the available integer and Boolean operations for brevity.

The grammar for this subset is given in Figure 6. The grammar for types, layout definitions and algebraic data type definitions remain the same as before and are therefore omitted. `instA,B(f)` corresponds to `instantiate [A] B f`. We also include another construct, `lowerA(C e1 ... en)`. This says to use the specific layout A for the given constructor application $C e_1 \dots e_n$.

The semantics for SSL are largely derived [9] from standard separation logic semantics. [11]

```

⟨i⟩ ::= ... | -2 | -1 | 0 | 1 | 2 | ...
⟨b⟩ ::= true | false
⟨e⟩ ::= ⟨var⟩ | ⟨i⟩ | ⟨b⟩ | ⟨e⟩ + ⟨e⟩ | C  $\overline{\langle e \rangle}$  | instA,B(f)(⟨e⟩) | lowerA(⟨e⟩)
⟨fn-def⟩ ::=  $\overline{\langle fn-case \rangle}$ 
⟨fn-case⟩ ::= f ⟨pattern⟩ := ⟨e⟩

```

■ **Figure 6** Grammar for restricted Pika subset.

3.1 Overview of the Two Interpretations

The soundness theorem will link the abstract machine semantics to the translation. In fact, the abstract machine semantics and the translation are similar to each other. For the abstract machine we manipulate *concrete* heaps, while for the translation we generate *symbolic* heaps.

Comparing the two further, there are two main points (beyond what we've already mentioned) where these two interpretations of Pika differ:

1. When we need to unfold a layout, how do we know which layout branch to choose?
2. How do we translate function applications (including, but not limited to, recursive applications)?

First, consider the abstract machine semantics. In this case, we are able to choose which branch of a layout to use by evaluating the expression we are applying it to until the expression is reduced to a constructor value (where a “constructor value” is either a value or a constructor applied to constructor values). If the expression is well-typed, this will always be a constructor of the algebraic data type corresponding to the layout. The two rules that this applies to are AM-LOWER and AM-INSTANTIATE. To interpret a function application, we interpret its arguments and substitute the results into the body of the function. We then proceed to interpret the substituted function body. This process is performed by the AM-INSTANTIATE rule.

Next, consider the SSL translation. Here we can determine which layout branch to use by generating a Boolean condition that will be true if and only if the SSL proposition on the right-hand side of the branch holds for the heap. Note that we assume that the programmer-supplied layout definitions are *injective* functions from algebraic data types to SSL assertions (up to bi-implication). We can directly translate function applications into SSL inductive predicate applications. Since inductive predicates already allow for recursive applications, there is no special handling necessary for recursion.

After defining these interpretations, we show how the abstract machine relation \rightarrowtail and the SSL interpretation function $\mathcal{T}[e]_{V,r}$ relate to each other by the Soundness Theorem 3.

3.2 Abstract Machine Semantics

In this section, we will define an abstract machine semantics for Pika and relate this to the standard semantics for SSL.

3.2.1 Notation and Setup

The set of values is $\text{Val} = \mathbb{Z} \cup \mathbb{B} \cup \text{Loc}$. Each of these three sets is disjoint from the other two. In particular, note that Loc and \mathbb{Z} are disjoint.

There is also a set of **Pika** values FsVal . This includes all the elements of Val , but also includes “constructor values” given by the rules in Figure 7. In addition to the store and heap of standard SSL semantics, the abstract machine semantics uses an FsStore . This is a partial function from locations to **Pika** values: $\text{FsStore} = \text{Loc} \rightarrow \text{FsVal}$. The primary purpose of this is to recover constructor values when given a location.

The general format of the transition relation is $(e, \sigma, h, \mathcal{F}) \mapsto (v, \sigma', h', \mathcal{F}', r)$, where the expression e results in the store being updated from σ to σ' , the heap being updated from h to h' , v is the Val obtained by evaluating e , the initial and final FsStore are \mathcal{F} and \mathcal{F}' and the result is stored in variable r . We assume that there is a global environment Σ which contains all layout definition equations and function definition equations.

Given a heap h and a heap layout H , we will make use of the notation $h \cdot [H]$. This extends the heap with the location assignments given in H . We say that the layout body H is *acting* on the heap h . This is defined in Figure 9. The intuition for this is that h gets updated using the symbolic heap description in H . For example, $\emptyset \cdot [a : \rightarrow 7]$ will contain only the value 7 at the location a . It is assumed that H does not have any variables on the right-hand side of $: \rightarrow$.

3.2.2 Abstract Machine Rules

The abstract machine semantics provides big-step operational semantics for evaluating **Pika** expressions on a heap machine. Its rules, given by Figure 8, make use of standard SSL models:

Model	$\mathcal{M} ::= (\sigma, h)$
Store	$\sigma : \text{Var} \rightarrow \text{Val}$
Heap	$h : \text{Loc} \rightarrow \text{Val}$

Note that a compound expression, consisting of multiple subexpressions, uses *disjoint* parts of the heap for each subexpression. This can be seen in the AM-ADD, AM-LOWER and AM-INSTANTIATE rules, which is essential for the proof of Soundness Theorem 3.

3.3 Translating Pika Specifications into SSL

We will define two translations: One from **Pika** *expressions* into SSL propositions and the other from **Pika** *definitions* into SSL inductive predicate definitions. We start with the former.

3.3.1 Translating Expressions

In the rules given in Figure 10, the notation $\mathcal{I}_{A,B}(f)$ gives the name of the inductive predicate that the **Pika** function f translates to when it is instantiated to the layouts A and B .

We start by defining the translation rules for expressions. We use these translation rules in Lemma 1 to define a translation function $\mathcal{T}[\cdot]_{V,r}$. Then, we will define translation rules for function definitions. The translation relation for expressions has the form $(e, V) \Downarrow (p, s, V', v)$, where p and s are the pure part and spatial part (respectively) of an SSL assertion and $V, V' \in \mathcal{P}(\text{Var})$.

The rules can be thought of as being in two groups:

1. Rules for base type expressions, such as S-LIT and S-ADD.
2. Rules for using layouts to translate expressions whose types involve algebraic data types.

Examples include S-LOWER-CONSTR and S-INST-INST.

$$\frac{x \in \text{Val}}{x \in \text{FsVal}} \text{ FsVal-BASE} \quad \frac{x_1 \in \text{FsVal} \cdots x_n \in \text{FsVal}}{(C x_1 \cdots x_n) \in \text{FsVal}} \text{ FsVal-CONSTR}$$

Figure 7 FsVal judgment rules.

$$\begin{array}{c} r \text{ fresh } i \in \mathbb{Z} \sigma' = \sigma \cup \{(r, i)\} \text{ AM-INT} \quad \frac{r \text{ fresh } b \in \mathbb{B} \sigma' = \sigma \cup \{(r, b)\}}{(b, \sigma, \emptyset, \mathcal{F}) \mapsto (b, \sigma', \emptyset, \mathcal{F}, r)} \text{ AM-BOOL} \\ v \in \text{dom}(\sigma) \sigma(v) \notin \text{Loc} \quad \frac{}{(v, \sigma, \emptyset, \mathcal{F}) \mapsto (\sigma(v), \sigma, \emptyset, \mathcal{F}, v)} \text{ AM-VAR-BASE} \\ v \in \text{dom}(\sigma) \sigma(v) \in \text{Loc} \quad \frac{}{(v, \sigma, \emptyset, \mathcal{F}) \mapsto (\mathcal{F}(\sigma(v)), \sigma, \emptyset, \mathcal{F}, v)} \text{ AM-VAR-LOC} \\ (x, \sigma, \mathcal{F}, h_1) \mapsto (x', \sigma_x, h'_1, \mathcal{F}, v_x) \quad (y, \sigma_x, \mathcal{F}, h_2) \mapsto (y', \sigma_y, h'_2, \mathcal{F}, v_y) \\ r \text{ fresh } h = h_1 \circ h_2 \quad h' = h'_1 \circ h'_2 \quad z = x' + y' \quad \sigma' = \sigma_y \cup \{(r, z)\} \quad \frac{}{(x + y, \sigma, h, \mathcal{F}) \mapsto (z, \sigma', h', \mathcal{F}, r)} \text{ AM-ADD} \\ (A[x] (C a_1 \cdots a_n) := H) \in \Sigma \quad (e, \sigma_0, h_0) \mapsto (C e_1 \cdots e_n, \sigma_1, h_1, y_1) \\ (e_i, \sigma_i, h_i) \mapsto (e'_i, \sigma_{i+1}, h'_i, v_i) \text{ for each } 1 \leq i \leq n \\ h' = h'_1 \circ h'_2 \circ \cdots \circ h'_n \quad h = h_0 \circ h_1 \circ \cdots \circ h_n \quad \sigma' = \sigma_{n+1} \cup \{(r, \ell)\} \\ \ell \text{ fresh } r \text{ fresh} \\ H' = H[x := \ell][a_1 := \sigma_2(v_1)][a_2 := \sigma_3(v_2)] \cdots [a_n := \sigma_{n+1}(v_n)] \\ \mathcal{F}' = \mathcal{F} \cup \{(\ell, (C e'_1 \cdots e'_n))\} \quad \frac{}{(\text{lower}_A(e), \sigma_0, h, \mathcal{F}) \mapsto ((C e'_1 \cdots e'_n), \sigma', h' \cdot [H'], \mathcal{F}', r)} \text{ AM-LOWER} \\ (A[x] (C a) := H) \in \Sigma \quad (f (C b) := e_f) \in \Sigma \\ (e, \sigma, h) \mapsto (C e_1, \sigma_1, h_1, y) \\ (e_1, \sigma_1, h_1) \mapsto (e'_1, \sigma_2, h_2, r) \\ \ell \text{ fresh } r \text{ fresh } y \text{ fresh} \\ H' = H[x := \ell][a := e'_1] \quad h' = h_1 \cdot [H'] \quad \sigma' = \sigma_f \cup \{(r, \ell)\} \\ \mathcal{F}' = \mathcal{F} \cup \{(\ell, (C e'_1))\} \\ \frac{(\text{lower}_B(e_f[b := y]), \sigma_2, h') \mapsto (e'_f, \sigma_f, h'', r)}{(\text{inst}_{A,B}(f)(e), \sigma, \mathcal{F}, h) \mapsto (e'_f, \sigma', h'', \mathcal{F}', r)} \text{ AM-INSTANTIATE} \end{array}$$

Figure 8 Abstract machine semantics rules.

$$\begin{array}{c} h \cdot [\text{emp}] = h \text{ L-EMP} \quad \frac{h' = h \cdot [H] \quad a \in \text{Val}}{h \cdot [\ell : \rightarrow a, H] = h'[\ell \mapsto a]} \text{ L-POINTSTO} \\ \frac{e \in \text{Val}}{h \cdot [A[x](e), H] = h \cdot [H]} \text{ L-APPLY} \end{array}$$

Figure 9 Rules for layout bodies acting on heaps.

In the first group, consider S-ADD. In the result of the translation, we've included $v == v_1 + v_2$ in the list of conjuncts in the pure part. Here, v_1 and v_2 are the results of the two subexpressions in the addition. In the pure part, we also include the pure parts of the two subexpressions as conjuncts. These are p_1 and p_2 . The spatial part of the translation consists of the spatial parts of the two subexpressions, s_1 and s_2 .

Now, in the second group, consider S-LOWER-CONSTR. This translates a Pika constructor application expression using a specific layout (which is provided by using the `lower_-` construct). It takes the specific branch of the layout corresponding to the constructor in question and puts the right-hand side of that branch into the spatial part, after applying the appropriate substitutions for the arguments given to the constructor in the application. The right-hand side of the layout branch is H and, after the substitution, it is called H' .

The S-INST-INST rule is used to translate a function application being applied to the result of another function application, given particular layouts for each application. In SuSLik, it does not make sense to directly apply a predicate to another predicate application. Therefore, we must do an ANF-like translation, where the result does not have “compound” applications like this. This translation is exactly what S-INST-INST is doing.

► **Lemma 1** ($\mathcal{T}[\cdot]$ function). $(\cdot, V) \Downarrow (\cdot, \cdot, \cdot, r)$ is a computable function $\text{Expr} \rightarrow (\text{Pure} \times \text{Spatial} \times \mathcal{P}(\text{Var}))$, given fixed V and r where $r \notin V$.

By throwing away the third element of the tuple in the codomain, we obtain a function $\text{Expr} \rightarrow (\text{Pure} \times \text{Spatial})$ from expressions to SSL propositions.

Call this function $\mathcal{T}[\cdot]_{V,r}$. That is, we define the function as follows where $r \notin V$:

$$\mathcal{T}[e]_{V,r} = (p; s) \iff (e, V) \Downarrow (p, s, V', r) \text{ for some } V'$$

We highlight the computability of this function to emphasise the fact that it can be used directly in an implementation of this subset of Pika.

3.3.2 Translating Function Definitions

The next step is to define the translation for Pika function definitions. In order to do this, we must first figure out how to determine the appropriate layout branch to use when unfolding a layout, a problem we highlighted earlier. Once this is accomplished, the rest of the translation can be defined. When this problem was solved for the abstract machine semantics, it was possible to simply evaluate the Pika expression until a constructor application expression was reached. From there, it is possible to just look at the constructor name and match it against the appropriate layout branch.

For the translation, however, we do not have the luxury of being able to evaluate expressions. Instead, we must instead rely on the fact that, in SSL, a “pure” (Boolean) condition can determine which inductive predicate branch to use. The question becomes: *Given an algebraic data type and a layout for that ADT, how do we generate an appropriate Boolean condition for a given constructor for the ADT?*

The solution is to find a Boolean condition which, given that the inductive predicate holds, is true if and only if the layout branch corresponding to that ADT constructor is satisfiable. In more detail, to define the branches of an inductive predicate $\mathcal{I}_A(x)$, given an ADT α , a constructor $C : \beta_1 \rightarrow \beta_2 \rightarrow \dots \rightarrow \beta_n \rightarrow \alpha$, a layout $A : \alpha \rightarrow \text{layout}[x]$ with a branch $A[x] (C a_1 \dots a_n) := H$ and given that $\mathcal{I}_A(x)$ holds, find a Boolean expression b with one free variable x such that $b \iff \exists \sigma, h. (\sigma, h) \models H$.

$$\begin{array}{c}
 \frac{i \in \mathbb{Z} \quad v \text{ fresh}}{(i, V) \Downarrow (v == i, \text{emp}, V \cup \{v\}, v)} \text{ S-INT} \\
 \frac{b \in \mathbb{B} \quad v \text{ fresh}}{(b, V) \Downarrow (v == b, \text{emp}, V \cup \{v\}, v)} \text{ S-BOOL} \\
 \frac{v \in \text{Var}}{(v, V) \Downarrow (\text{true}, \text{emp}, V, v)} \text{ S-VAR} \\
 \frac{(e_1, V_0) \Downarrow (p_1, s_2, V_1, v_1) \quad (e_2, V_1) \Downarrow (p_2, s_2, V_2, v_2) \quad v \text{ fresh}}{(e_1 + e_2, V_0) \Downarrow (v == v_1 + v_2 \wedge p_1 \wedge p_2, s_1 * s_2, V_2 \cup \{v\}, v)} \text{ S-ADD} \\
 \frac{v \in \text{Var}}{(\text{lower}_A(v), V) \Downarrow (\text{true}, A(v), V \cup \{v\}, v)} \text{ S-LOWER-VAR} \\
 \frac{(A[x] (C a_1 \cdots a_n) := H) \in \Sigma \quad (e_i, V_i) \Downarrow (p_i, s_i, V_{i+1}, v_i) \text{ for each } 1 \leq i \leq n \quad v \text{ fresh}}{V' = V_{n+1} \cup \{x\} \quad H' = H[a_1 := v_1] \cdots [a_n := v_n] \quad (\text{lower}_A(C e_1 \cdots e_n), V_1) \Downarrow (p_1 \wedge \cdots \wedge p_n, H' * s_1 * \cdots * s_n, V', x)} \text{ S-LOWER-CONSTR} \\
 \frac{v \in V \quad r \text{ fresh}}{(\text{inst}_{A,B}(f)(v), V) \Downarrow (\text{true}, \mathcal{I}_{A,B}(f)(v, r), V \cup \{r\}, r)} \text{ S-INST-VAR} \\
 \frac{(A[x] (C a_1 \cdots a_n) := H) \in \Sigma \quad (e_i, V_i) \Downarrow (p_i, s_i, V_{i+1}, v_i) \text{ for each } 1 \leq i \leq n \quad x \text{ fresh}}{V' = V_{n+1} \cup \{x\} \quad H' = H[a_1 := v_1] \cdots [a_n := v_n] \quad (f (C b_1 \cdots b_n) := e_f) \in \Sigma \quad e'_f = e_f[b_1 := v_1] \cdots [b_n := v_n] \quad (\text{lower}_B(e'_f), V_{n+1}) \Downarrow (p, s, V', r)} \text{ S-INST-CONSTR} \\
 \frac{(\text{inst}_{A,B}(g)(e), V) \Downarrow (p_1, s_1, V_1, r_1) \quad (\text{inst}_{B,C}(f)(r_1), V_1) \Downarrow (p_2, s_2, V_2, r_2)}{(\text{inst}_{B,C}(f)(\text{inst}_{A,B}(g)(e)), V) \Downarrow (p_1 \wedge p_2, s_1 * s_2, V_2, r_2)} \text{ S-INST-INST}
 \end{array}$$

Figure 10 Expression Translation Rules.

$V = \{v_1, \dots, v_n\}$ where v_1, \dots, v_n are distinct variables

$r \in \text{Var} \quad r \notin V \quad c = \text{cond}(A, C, x) \quad p_1 \cdots p_n \text{ fresh}$

$$\frac{(p, s) = \mathcal{T}[\text{inst}_{A,B}(f)(C p_1 \cdots p_n)]_{V,r} \quad (f (C a_1 \cdots a_n) := e) \xrightarrow{\text{fn-def}}_{A,B} (\mathcal{I}_{A,B}(f)(x, r) : c \Rightarrow \{p; s\})}{(f (C a_1 \cdots a_n) := e) \xrightarrow{\text{fn-def}}_{A,B} (\mathcal{I}_{A,B}(f)(x, r) : c \Rightarrow \{p; s\})} \text{ FNDEF}$$

Figure 11 Translation rule for function definitions.

► **Lemma 2** (cond function). *There is a computable function $\text{cond}(\cdot, \cdot)$ that takes in any layout $A : \alpha \rightarrow \text{layout}[x]$ with a branch $A[x]$ ($C a_1 \cdots a_n := H$ for a given constructor $C : \beta_1 \rightarrow \beta_2 \rightarrow \cdots \rightarrow \beta_n \rightarrow \alpha$ and it produces a Boolean expression with one free variable x such that the following holds under the assumption that $\mathcal{I}_A(x)$ holds.*

$$\text{cond}(A, C) \iff \exists \sigma, h. (\sigma, h) \models H$$

Here, $\mathcal{I}_A(x)$ is the name of the generated inductive predicate corresponding to the layout A .

With this function in hand, we are now ready to define the translation for Pika function definitions. This definition is in Figure 11. In this rule, the fresh variables p_1, \dots, p_n will be substituted for a_1, \dots, a_n .

3.4 Typing Rules

Typing rules for Pika expressions are given in Figure 12. These rules differ from standard typing rules for a functional language due to the existence of layouts and their associated constructors, like `instantiate` and `lower`. If an expression is well-typed, then each use of `instantiate` and `lower` only uses layouts together with the ADT that they are defined for.

The rules also make use of a *concreteness judgment*. The rules for this judgment are given in Figure 13a. The intuition of this judgment is that a type is “concrete” iff values of that type can be directly represented in the heap machine semantics. For example, an ADT type is *not* concrete because a layout has not been specified. However, once a particular layout is specified for the ADT type, it becomes concrete. Base types, like `Int`, are also concrete.

Rules for the ensuring that global definitions are well-typed are given in Figure 13b. In this figure, Δ is the set of all (global) constructor type definitions.

3.5 From Pika to SLL Specifications: Soundness of the Translation

We want to show that our abstract machine semantics and our SLL translation fit together. In particular, our abstract machine semantics should generate models that satisfy the separation logic propositions given by our SLL translation. Figure 14 gives a high-level overview of how these pieces fit together. We will give a more specific description of this in Theorem 3.

► **Theorem 3** (Soundness). *For any well-typed expression e , if $\mathcal{T}\llbracket e \rrbracket_{V,r}$ is satisfiable for $V = \text{dom}(\sigma')$ and $(e, \sigma, h, \mathcal{F}) \mapsto (e', \sigma', h', \mathcal{F}', r)$, then $(\sigma', h') \models \mathcal{T}\llbracket e \rrbracket_{V,r}$. That is, given an expression e with a satisfiable SLL translation, any heap machine state that e transitions to (by the abstract machine semantics) will be a model for the SLL translation of e (cf. Figure 14).*

Proof. See the Appendices in the extended version of the paper [14]. ◀

The fact that, at the top level, we only translate function definitions suggests an additional theorem. We want to specifically show that any possible function application is sound, in the sense just described. This immediately follows from Theorem 3.

Abbreviating $\text{inst}_{A,B}(f)$ as $f_{A,B}$, we can state the following theorem:

► **Theorem 4** (Application soundness). *For any well-typed function application $f_{A,B}(e)$, if $\mathcal{T}\llbracket f_{A,B}(e) \rrbracket_{V,r}$ is satisfiable for $V = \text{dom}(\sigma')$ and $(f_{A,B}(e), \sigma, h, \mathcal{F}) \mapsto (e', \sigma', h', \mathcal{F}', r)$, then $(\sigma', h') \models \mathcal{T}\llbracket f_{A,B}(e) \rrbracket_{V,r}$.*

Proof. This follows immediately from Theorem 3. ◀

$$\begin{array}{c}
 \frac{i \in \mathbb{Z}}{\Gamma \vdash i : \text{Int}} \text{-INT} \quad \frac{b \in \mathbb{B}}{\Gamma \vdash b : \text{Bool}} \text{-BOOL} \quad \frac{(v : \alpha) \in \Gamma}{\Gamma \vdash v : \alpha} \text{-VAR} \\
 \\
 \frac{(f : \alpha \rightarrow \beta) \in \Sigma}{\Gamma \vdash f : \alpha \rightarrow \beta} \text{-FN-GLOBAL} \\
 \\
 \frac{\Gamma \vdash x : \text{Int} \quad \Gamma \vdash y : \text{Int}}{\Gamma \vdash x + y : \text{Int}} \text{-ADD} \\
 \\
 \frac{(v : \alpha) \in \Gamma \quad (A : \alpha \rightsquigarrow \text{layout}[x]) \in \Sigma}{\Gamma \vdash \text{lower}_A(v) : A} \text{-LOWER-VAR} \\
 \\
 \frac{(C : \alpha_1 \rightarrow \dots \rightarrow \alpha_n \rightarrow \beta) \in \Sigma \quad (B : \beta \rightsquigarrow \text{layout}[x]) \in \Sigma}{\Gamma \vdash e_i \text{ concrete}_{\alpha_i} \text{ for each } i \text{ with } 1 \leq i \leq n} \text{-LOWER-CONSTR} \\
 \\
 \frac{(A : \alpha \rightsquigarrow \text{layout}[x]) \in \Sigma \quad (B : \beta \rightsquigarrow \text{layout}[y]) \in \Sigma}{\Gamma \vdash f : \alpha \rightarrow \beta \quad \Gamma \vdash e : A} \text{-INSTANTIATE} \\
 \\
 \frac{(C : \alpha_1 \rightarrow \dots \rightarrow \alpha_n \rightarrow \beta) \in \Sigma \quad \Gamma \vdash e_i : \alpha_i \text{ for each } i \text{ with } 1 \leq i \leq n}{\Gamma \vdash C e_1 \dots e_n : \beta} \text{-CONSTR}
 \end{array}$$

Figure 12 Typing rules.

4 Extensions of SuSLik

We have shown the translation from the functional specifications into SSL specifications. However, some of the SSL specifications are not supported in the original SuSLik and existing variants. In this section, we show how to extend the SuSLik to support more features to make the whole thing work. We will show the extensions on the following three aspects:

- How to describe and call an existing function within SSL predicates.
- How to make the result of one function call as the input of another function call.
- How to synthesise programs with inductive predicates without the help of pure theory.

$$\begin{array}{c}
 \frac{\Gamma \vdash e : \text{Int}}{e \text{ concrete}_{\text{Int}}} \text{C-INT} \quad \frac{\Gamma \vdash e : \text{Bool}}{e \text{ concrete}_{\text{Bool}}} \text{C-BOOL} \\
 \\
 \frac{(A : \alpha \rightsquigarrow \text{layout}[x]) \in \Sigma \quad \Gamma \vdash e : A}{e \text{ concrete}_{\alpha}} \text{C-LAYOUT} \quad \frac{\begin{array}{c} (C : \alpha_1 \rightarrow \alpha_2 \rightarrow \dots \rightarrow \alpha_n \rightarrow \beta) \in \Delta \\ b_1 : \alpha_1, b_2 : \alpha_2, \dots, b_n : \alpha_n \vdash e : \gamma \end{array}}{(f(C b_1 \dots b_n) := e) \Rightarrow f : \beta \rightarrow \gamma} \text{G-FN}
 \end{array}$$

(a) Concreteness judgment rules.

(b) Global definition typing.

Figure 13 Rules for concreteness judgement and typing global definitions.

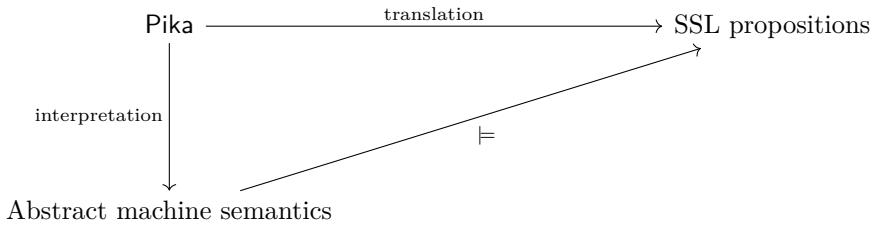


Figure 14 The relationship between the two Pika semantics given by the soundness theorem.

4.1 Function Predicates

Without any modification upon the implementation, we find the SSL predicate with some restrictions can be used to describe function relations other than data structures (named function predicates). The definition of **function predicates** is as follows:

► **Definition 5 (Function Predicates).** Given any non-higher-order n-ary function $f(x_1, \dots, x_n)$ in the functional language, the function predicate to synthesise f has the following format:

```
predicate predf(T x1, ... ,T xn, T output){...}
```

where $T \in \{\text{loc}, \text{int}\}$. The type of x_i (and output) is decided by the type of f . If it is an integer in f , then its type is int ; otherwise, it is loc (for any data structure in Pika).

Since the input of the whole workflow is functional programs, the “output” in the definition is to provide another location for the output of the function. And the specification to synthesise function f should have the following format:

```
void f(loc x1, ... ,loc xn)
{x1 :-> v1 ** x2 :-> 12 ** s11(12) ** ... ** xn :-> vn ** output :-> 0}
{x1 :-> v1 ** x2 :-> 12 ** ... ** xn :-> vn ** output :-> output0 **
 predf(v1, 12, ..., vn, output0)}
```

4.2 SSL Rules for *func* Structure

As we show in previous examples, the reason we can have *func* structure is that the points-to structure in the post-condition is always eliminated after write operations. For example, the in-placed *inc1* functions specification is satisfied via the WRITE (Figure 15) on the location.

```
void inc_y(loc y, loc x)
{x :-> vx ** y :-> vy}
{x :-> vx + vy ** y :-> vy}
```

The core insight of *func* structure is: since the function synthesised by function predicate behaves like the pure function, it is the same as the WRITE rule in the sense that only the output location is modified. Thus, we add the new FUNCWRITE rule into the zoo of SSL rules (see Figure 15). To make the *func* structure correctly equal to some “write” operation, the following restrictions should hold, which are achieved by the translation:

- If $\text{func } f(x_1, \dots, x_n, \text{output})$ appears in a post-condition, then no write rule can be applied to any x_i . This is to avoid the ambiguity of the *func*.
- The type of function f is consistent.

$$\begin{array}{c}
 \text{WRITE} \\
 \frac{\text{Vars}(e) \subseteq \Gamma \quad e \neq e' \quad \{\phi; x \mapsto e * P\} \rightsquigarrow \{\psi; x \mapsto e * Q\} | c}{\{\phi; x \mapsto e' * P\} \rightsquigarrow \{\psi; x \mapsto e * Q\} | *x = e ; c} \\
 \\
 \text{FUNCWRITE} \\
 \frac{\forall i \in [1, n], \text{Vars}(e_i) \subseteq \Gamma \quad \{\phi; P\} \rightsquigarrow \{\psi; Q\} | c}{\{\phi; x \mapsto e * P\} \rightsquigarrow \{\psi; \text{func } f(e_1, \dots, e_n, x) * Q\} | f(e_1, \dots, e_n, x) ; c}
 \end{array}$$

Figure 15 The WRITE and new FUNCWRITE rules in SSL.

Note that based on the setting of the function predicate, the parameters of the function call are pointers, while the parameters of the function predicate are content to which pointers point. Furthermore, we have the *func* generated from function predicates and with the format defined in Subsection 4.1. As a result, the equivalent original SSL that duplicates points-to of one location is not a problem, since they can be merged as one.

4.3 Temporary Location for the Sequential Application

Though richly expressive, SSL has difficulty in expressing the sequential application of functions. For example, given the *func* structure available, the following function is not expressible within one function predicate:

```
f x y = g (h x) y
```

If we attempt to express it, we will have the following part in the predicate:

```
predicate f(loc x, loc y, loc output)
{... ** func h(x, houtput) ** func g(houtput, y, output)}
```

However, *houtput* is not a location in the pre-condition, which is not allowed in SSL. Thus, we introduce a new keyword *temp* to denote the temporary location for the sequential application. The new definition of *func* is as follows:

```
predicate f(loc x, loc y, loc output)
{... ** temp houtput ** func h(x, houtput) ** func g(houtput, y, output)}
```

Roughly speaking, the *temp* structure will help to allocate a new location for the output of the first function, and then use it as the input of the second function. After all appearances of *houtput* is eliminated, we will deallocate the location.

Note that the temporary variable is possible to appear in two different structures: recursive function predicates or *func* call. The reason we don't need to consider the basic arithmetic operations is that the integer will be directly used as the predicate parameter, instead of the location as the parameter. For example, the sum of a list can be expressed as:

```
predicate sum(loc l, int output){
| l == 0 => {output == 0; emp}
| l != 0 => {output == output1 + v; [l, 2] ** l :-> v ** l + 1 :-> lnxt **
sum(lnxt, output1)} }
```

Such sequential application is common in functional programming, especially in the recursive function. For example, it is not elegant to flatten a list of lists without the sequential application.

$$\text{TEMPFUNCALLOC} \quad \frac{\{\phi; x \mapsto a * P\} \rightsquigarrow \{\psi; \text{func } f(e_1, \dots, e_n, x) * \text{temp}(x, 1) * Q\} | c}{\{\phi; P\} \rightsquigarrow \{\psi; \text{func } f(e_1, \dots, e_n, x) * \text{temp}(x, 0) * Q\} | \text{let } x = \text{malloc}(1); c}$$

$$\text{TEMPFUNCFREE} \quad \frac{\{\phi; P\} \rightsquigarrow \{\psi; Q\} | c}{\exists x \in Q \wedge \{\phi; P\} \rightsquigarrow \{\psi; \text{temp}(x, 1) * Q\} | \text{let } x_0 = *x; \text{type_free}(x_0); \text{free}(x); c}$$

■ **Figure 16** New allocating and deallocating rule for `temp` in SSL.

```
flatten :: [[a]] -> [a]
flatten [] = []
flatten (x:xs) = x ++ flatten xs
```

We can express this function, but with some strange structure to store all temporary lists.

```
predicate flatten(loc x, loc output){
| x == 0 => {output :-> 0}
| x != 0 => {[x, 2] ** x :-> x0 ** sll(x0) ** x + 1 :-> xnxt **
[output, 2] ** func append(x, outputnxt, output) ** output + 1 :-> outputnxt **
flatten(xnxt, outputnxt)} }
```

With such a function predicate, though we can synthesise the function whose result stored in `output` is the flattened list, the list `output` is containing a lot of intermediate values, which is neither consistent with the definition in the source language nor space efficient.

The new rules consist of allocating and deallocating rules (Figure 16). Based on the definition of the `func` structure and the function predicate, the allocated locations are different, where the `temp` location for `func` is directly used; while the `temp` location for function predicate should allocate a new location for function predicates. As for the deallocation, not only the `temp` location(s) but also the content they point to should be deallocated. That is the reason we have the `type_free` function, which is syntax sugar to deallocate the content of a location based on the type information. For example, if the type of the location is `tree`, then the `type_free` will deallocate the content of the location via `tree_free` function, which is synthesised based on the SSL predicate `tree` as follows.

```
void tree_free(loc x)
{tree(x)}
{emp}
```

Specifically, if the location contains the value with type `int`, then the `type_free` will do nothing. Thus, the function predicate with `temp` is much better, in the sense that no extra space is used, and the synthesised function is consistent with the source language.

```
predicate flatten(loc x, loc output){
| x == 0 => {output :-> 0}
| x != 0 => {[x, 2] ** x :-> x0 ** sll(x0) ** x + 1 :-> xnxt ** temp outputnxt **
** flatten(xnxt, outputnxt) ** func append(x, outputnxt, output)} }
```

4.4 Avoiding Excessive Heap Manipulation with Read-Only Locations

The existing SuSLik depends on the set theory to express the pure relation. However, it is not trivial to automatically generate the pure part of SSL specifications from the functional specifications. To see why the set theory is needed, the following simple example shows the functionality of the set theory, with `sll_n` being the single-linked list with `no` set.

```

predicate sll_n(loc x) {
| x == 0      => {true; emp}
| not (x == 0) => { [x, 2] ** x :-> v ** (x + 1) :-> xnxt ** sll(xnxt) } }
predicate copy(loc x, loc y) {
| x == 0      => {y == 0; emp}
| not (x == 0) => { [y, 2] ** y :-> v ** (y + 1) :-> ynxt ** [x, 2] ** x :-> v
                     ** (x + 1) :-> xnxt ** copy(xnxt, ynxt) } }

```

While the intent of the function predicate `copy` is to copy the list `x` to `y`, without the set theory, the output program will be somewhat surprising to see:

```

{sll_n(x)}
void copy (loc x, loc y) {
  if (x == 0) {
  } else {
    let n = *(x + 1); copy(n, y); let y01 = *y; let y0 = malloc(2); *y = y0;
    *(y0 + 1) = y01; let vy = *y0 *x = vy; } }
{copy(x, y)}

```

The problem here is that, when we have the pure relation in the predicate to indicate that the values are the same, the synthesiser finds another possible way: instead of copying the value of `x` to `y`, we can just change the value of `x` to initial value `vy` after `malloc`. This is not the user intent, and the output program is not correct. Turns out, the solution is not that difficult: we simply need add a new kind of heaplet in the specification language, call *constant points-to*, which has a similar idea as read-only borrows [2]. The only difference of the constant points-to from the original *points-to* heaplet is that the value of the location is constant, which means that the WRITE rule in SSL is not applicable. By this way, the extended SuSLik will not consider the modification of the input location, thus provides the correctness mechanism (in Subsection 3.5) for the translation of Pika.

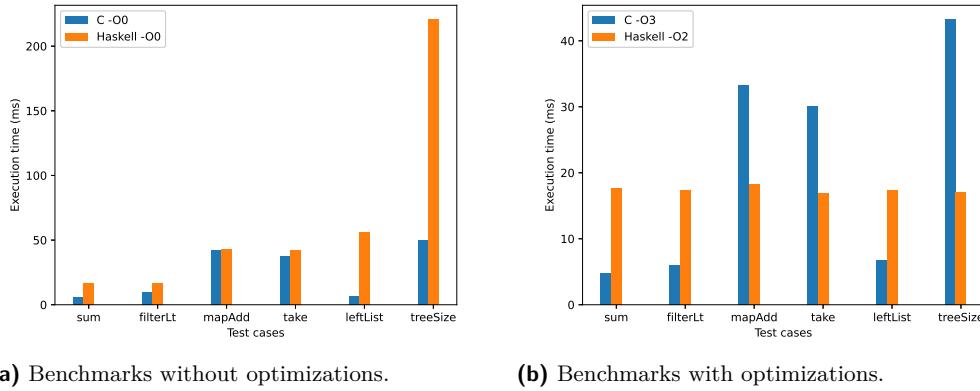
5 Evaluation

In this section, we evaluate Pika’s expressiveness. A secondary objective is to evaluate Pika’s performance. The performance evaluation is done largely to put Pika into context by comparing it to a prominent functional programming language (Haskell). The main purpose of Pika is to increase the expressiveness of SuSLik, which is the reason for the primary evaluation objective. Towards these goals, we answer the following research questions:

- **RQ1:** Is the performance of the synthesised code competitive with code generated from traditional functional language compilers?
- **RQ2:** In concrete terms, how does the tool’s expressivity compare with the expressivity of SuSLik specifications for programs written in a functional style?
- **RQ3:** What are the failure modes of our approach?

Our implementation and benchmarks are available in supplementary material. The experiments were conducted on a 2021 MacBook Pro with an M1 processor and 32 GB of RAM. We used GHC version 9.8.1 and Apple Clang version 13.0.0.

For **RQ1**, we run benchmarks and compare execution time. The benchmarks we selected are a series of functions that manipulate data structures such as lists and trees, which covers different common abstract operations (like `map`, `filter`, `fold`). The comparison is between Haskell functions based on user-defined data structures and C functions generated from Pika’s specifications (via the extended SuSLik), parameterised with the data structures of large size. We recorded both execution time with and without optimisation of compilers.



(a) Benchmarks without optimizations. (b) Benchmarks with optimizations.

Figure 17 Performance of C functions generated by Pika compared to Haskell/GHC.**Table 1** Statistics on the benchmarks: Pika spec size v, generated SSL spec size, translator performance, and synthesiser performance. All times are in seconds.

#	Task Name	Pika AST	SuSLik AST	$\frac{ \text{Pika AST} }{ \text{SuSLik AST} }$	Compile Time	Synthesis Time
1	cons	76	123	0.618	0.012	5.134
2	plus	46	91	0.505	0.002	5.992
3	add1Head	64	109	0.587	0.006	5.114
4	listId	62	107	0.579	0.005	4.906
5	add1HeadDLL	74	146	0.507	0.007	10.618
6	even	19	42	0.452	0.001	3.999
7	foldr	63	113	0.558	0.005	5.454
8	sum	58	103	0.563	0.004	5.125
9	filterLt	84	147	0.571	0.009	6.104
10	mapAdd	71	116	0.612	0.007	5.031
11	leftList	116	156	0.744	0.008	7.722
12	treeSize	78	126	0.619	0.007	5.980
13	take	119	212	0.561	0.012	10.447

The results are shown in Figure 17. Our findings are as follows:

- When compiled to an executable run without optimisations, the C programs generated by Pika are faster than Haskell programs. And we can also observe that the speed difference is larger for functions with more complex data structures.
- When compiled to an executable with optimisations:
 - For functions with complex data structures, the comparison is similar to the case without optimisation.
 - For functions for the singly-linked list GHC's optimisation is very powerful, resulting in much better performance than C programs generated by Pika. With more tests, we found out the performance after GHC's optimisation is similar to the one using Haskell's built-in list. We believe this is because GHC's optimisations are fine-tuned to optimise code manipulating list-like data structures and use the same optimisation as for Haskell's built-in list. That said, similar observations regarding GHC do not hold on other complex data structures, such as trees.

```

selfAppend : List -> List;
selfAppend xs := instantiate [S11, S11] S11 append xs xs;

append : List -> List -> List;
append (Nil) ys := ys;
append (Cons x xs) ys := instantiate [Int, S11[mutable]] S11 cons
                           (addr x) (append xs ys);

```

 **Figure 18** A Pika specification not supported by SuSLik.

To answer **RQ2**, we first find some common patterns for Pika programs in the benchmarks shown in Table 1: (1) pattern matching on ADTs (#9, 10, etc.), (2) code reusability (#3 vs 5, #7 vs 8). Those features are not directly expressible in SuSLik because of the low-level nature of SSL. For example, the `add1Head` and `add1HeadDLL` functions share the same function definition, where the only difference is the type layout used; but the SuSLik specifications need to be treated separately, which makes the codes more complex. To make some objective observations on the expressivity, we measure the number of nodes in the input Pika AST and find it consistently fewer than the number of AST nodes in the generated SuSLik specification.

To address **RQ3**, note that a particular failure mode occurs when Pika source code reuses a variable in a way that violates SSL constraints. For example, see the `selfAppend` example in Figure 18 which uses its argument twice. We could have addressed this issue by introducing a lightweight linear type system to Pika, but have not carried out this exercise yet. Another kind of failure occurs when SuSLik fails to synthesise an implementation of the generated specification: handling these failures is beyond the scope of this work.

6 Discussion

We have given a translation from a high-level functional language into SSL specifications to be given to a program synthesiser.

In doing so, we have revealed a close connection between, on one hand, algebraic data types and recursive pattern-matching functions and, on the other hand, SSL inductive predicates. The soundness of this connection is demonstrated by Theorem 3 in Section 3.

Beyond the theory, this connection can be exploited in three directions:

- Increased type safety: Algebraic data types allow you to distinguish between types that have the same runtime heap representation.
- More reusability: An algebraic data type can have multiple layouts, each of which gives a different possible runtime heap representation for the ADT. As Pika functions are defined only in terms of algebraic data types, they naturally get the *polymorphism* of the ADT by being able to work with any layout of the ADT.
- Greater succinctness: When working at this higher level of abstraction, it generally takes less code as you are not frequently manipulating heap locations. The only mention of heap locations is in reusable layout definitions. This can also give greater clarity.

7 Related Work

Pika is built upon the SuSLik synthesis framework. SuSLik provides a synthesis mechanism for heap-manipulating programs using a variant of Separation Logic [9]. However, it does not have any high-level abstractions. In particular, writing SuSLik specifications directly involves a significant amount of pointer manipulation. Further, it does not provide abstraction over specific memory layouts. As described in Subsection 2.2, Pika addresses these limitations.

Dargent language [1] also includes a notion of layouts and layout polymorphism for a class of algebraic data types, which differs from our treatment of layouts in two primary ways:

1. In **Pika**, abstract memory locations (with offsets) are used. In contrast, **Dargent** uses offsets that are all relative to a single “beginning” memory location. The **Pika** approach is more amenable to heap allocation, though this requires a separate memory manager of some kind. This is exposed in the generated language with `malloc` and `free`. On the other hand, the technique taken by **Dargent** allows for greater control over memory management. This makes dealing with memory more complex for the programmer, but it is no longer necessary to have a separate memory manager.
2. Algebraic data types in the present language include *recursive* types and, as a result, **Pika** has recursive layouts for these ADTs. This feature is not currently available in **Dargent**.

Furthermore, layout polymorphism also works differently. While **Dargent** tracks layout instantiations at a type-level with type variables, in the present work we simply only check to see if a layout is valid for a given type when type-checking. In particular, we cannot write type signatures that *require* the same layout in multiple parts of the type (for instance, in a function type `List -> List` we have no way at the type-level of requiring that the argument `List` layout and the result `List` layout are the same). This more rudimentary approach that **Pika** currently takes could be extended in future work. Overall, the examples in the **Dargent** paper tend to focus on the manipulation of integer values. In contrast, we have focused largely on data structure manipulation, which follow the primary motivation of **SuSLik**.

Synquid is another synthesis framework with a functional surface language. While **Synquid** allows an even higher-level program specification than **Pika** through its liquid types, it does not provide any access to low-level data structure representation. [8] In contrast, **Pika**’s level of abstraction is similar to that of a traditional functional language but, similar to **Dargent**, it also allows control over the data structure representation in memory.

8 Future Work and Conclusion

We make the following observations based on our experience of developing and using **Pika**.

By allowing layouts to use multiple SSL parameters, we would be able to give a greater variety of layouts associated with an ADT. For instance, the `List` data type used in the examples could have a doubly-linked list layout in addition to the singly-linked list layout `S11`. Note that any existing **Pika** function defined over `List` will continue to work with no modification with these new layouts. Defunctionalisation and lambda lifting can also be used to implement true higher-order functions.

It is possible to do inference of some layouts, for example in `mapAdd1` we would usually want to use the same layout as the argument layout, but we leave this for future work. Another approach is to introduce type variables that correspond to layouts, as done in the series of works on the **Dargent** tool [1]. We leave this approach for future work as well.

Reverse transformation deserves further investigation: if we go from an SSL specification to **Pika** program and then compile to, *e.g.*, C, can we synthesise additional programs that traditional SSL synthesisers would struggle with? What are the limitations of this approach?

We may be able to expose more of the synthesis mechanism in the **Pika** language. For example, generate an SSL specification given only a **Pika** type signature (and corresponding `%generate` directive). This could combine well with additional polymorphism, as we could utilise the free theorems that are given by a polymorphic type signature to further constrain the resulting specification.

Finally, is it possible to derive translations for languages such as Pika from abstract machine semantics? In this paper, we have given a language with abstract machine semantics. We then give a translation of that language into SSL. We then show that the final states given by the abstract machine semantics are models for the SSL propositions produced by our translation. But is it possible to begin by specifying the abstract machine semantics and then mathematically (or automatically) *derive* an appropriate translation into SSL, with the requirement that the translation satisfies the soundness theorem?

In conclusion, we have presented Pika: a high-level functional specification language that paves the way for the efficient synthesis of a verifiably correct imperative code with in-place memory updates that is comparable in efficiency to the handwritten C.

References

- 1 Zilin Chen, Ambroise Lafont, Liam O'Connor, Gabriele Keller, Craig McLaughlin, Vincent Jackson, and Christine Rizkallah. Dargent: A silver bullet for verified data layout refinement. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. doi:[10.1145/3571240](https://doi.org/10.1145/3571240).
- 2 Andreea Costea, Amy Zhu, Nadia Polikarpova, and Ilya Sergey. Concise Read-Only Specifications for Better Synthesis of Programs with Pointers. In *ESOP*, volume 12075 of *LNCS*, pages 141–168. Springer, 2020. doi:[10.1007/978-3-030-44914-8_6](https://doi.org/10.1007/978-3-030-44914-8_6).
- 3 Cordelia V. Hall, Kevin Hammond, Will Partain, Simon L. Peyton Jones, and Philip Wadler. The Glasgow Haskell Compiler: A Retrospective. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, Workshops in Computing, pages 62–71. Springer, 1992. doi:[10.1007/978-1-4471-3215-8_6](https://doi.org/10.1007/978-1-4471-3215-8_6).
- 4 C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. doi:[10.1145/363235.363259](https://doi.org/10.1145/363235.363259).
- 5 Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A History of Haskell: Being Lazy with Class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, pages 12–1–12–55. ACM, 2007. doi:[10.1145/1238844.1238856](https://doi.org/10.1145/1238844.1238856).
- 6 Shachar Itzhaky, Hila Peleg, Nadia Polikarpova, Reuben N. S. Rowe, and Ilya Sergey. Cyclic program synthesis. In *PLDI*, pages 944–959. ACM, 2021. doi:[10.1145/3453483.3454087](https://doi.org/10.1145/3453483.3454087).
- 7 Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local Reasoning about Programs that Alter Data Structures. In *CSL*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001. doi:[10.1007/3-540-44802-0_1](https://doi.org/10.1007/3-540-44802-0_1).
- 8 Nadia Polikarpova, Ivan Kuraç, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *PLDI*, pages 522–538. ACM, 2016. doi:[10.1145/2908080.2908093](https://doi.org/10.1145/2908080.2908093).
- 9 Nadia Polikarpova and Ilya Sergey. Structuring the synthesis of heap-manipulating programs. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:[10.1145/3290385](https://doi.org/10.1145/3290385).
- 10 John C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS*, pages 55–74, 2002. doi:[10.1109/LICS.2002.1029817](https://doi.org/10.1109/LICS.2002.1029817).
- 11 Reuben N. S. Rowe and James Brotherston. Automatic cyclic termination proofs for recursive procedures in separation logic. In *CPP*, pages 53–65. ACM, 2017. doi:[10.1145/3018610.3018623](https://doi.org/10.1145/3018610.3018623).
- 12 David Turner. An Overview of Miranda. *SIGPLAN Not.*, 21(12):158–166, 1986. doi:[10.1145/15042.15053](https://doi.org/10.1145/15042.15053).
- 13 Yasunari Watanabe, Kiran Gopinathan, George Pîrlea, Nadia Polikarpova, and Ilya Sergey. Certifying the synthesis of heap-manipulating programs. *Proc. ACM Program. Lang.*, 5(ICFP):1–29, 2021. doi:[10.1145/3473589](https://doi.org/10.1145/3473589).
- 14 David Young, Ziyi Yang, Ilya Sergey, and Alex Potanin. Higher-Order Specifications for Deductive Synthesis of Programs with Pointers (Extended Version). *CoRR*, abs/2407.09143, 2024. doi:[10.48550/arXiv.2407.09143](https://doi.org/10.48550/arXiv.2407.09143).

CtChecker: A Precise, Sound and Efficient Static Analysis for Constant-Time Programming

Quan Zhou  

Penn State University, University Park, PA, USA

Sixuan Dang  

Duke University, Durham, NC, USA

Danfeng Zhang  

Duke University, Durham, NC, USA

Abstract

Timing channel attacks are emerging as real-world threats to computer security. In cryptographic systems, an effective countermeasure against timing attacks is the constant-time programming discipline. However, strictly enforcing the discipline manually is both time-consuming and error-prone. While various tools exist for analyzing/verifying constant-time programs, they sacrifice at least one feature among precision, soundness and efficiency.

In this paper, we build CtChecker, a sound static analysis for constant-time programming. Under the hood, CtChecker uses a static information flow analysis to identify violations of constant-time discipline. Despite the common wisdom that sound, static information flow analysis lacks precision for real-world applications, we show that by enabling field-sensitivity, context-sensitivity and partial flow-sensitivity, CtChecker reports fewer false positives compared with existing sound tools. Evaluation on real-world cryptographic systems shows that CtChecker analyzes 24K lines of source code in under one minute. Moreover, CtChecker reveals that some repaired code generated by program rewriters supposedly remove timing channels are still not constant-time.

2012 ACM Subject Classification Security and privacy → Information flow control

Keywords and phrases Information flow control, static analysis, side channel, constant-time programming

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.46

Supplementary Material Software (*ECOOP 2024 Artifact Evaluation approved artifact*):
<https://doi.org/10.4230/DARTS.10.2.26>

Funding This work was supported by the NSF under grants 2401182, 2401496, 1942851 and 1956032.

Acknowledgements We express our sincere gratitude to the anonymous reviewers for their insightful feedback and suggestions. We would like to thank Shuai Wang for sharing detailed CacheS evaluation results, and Ernest DeFoy III and Xiang Li for their contributions in the early stage of the project.

1 Introduction

Modern cryptographic systems are vulnerable to timing attacks [21, 10, 26, 6], which can quickly reveal confidential keys by analyzing the encryption/decryption time of those systems. While the threat has been well-known for decades, identifying timing channel vulnerabilities in cryptographic systems is a daunting task, as timing channels result from implementation details such as data and instruction cache effects, branch prediction buffers and memory controllers. As all of those hardware features are invisible in the source code, manually identifying timing channel vulnerabilities is extremely challenging, if possible at all, as doing so precisely requires a crystal clear view of the whole software-hardware stack and how secret information flows throughout the stack.

 © Quan Zhou, Sixuan Dang, and Danfeng Zhang;
licensed under Creative Commons License CC-BY 4.0
38th European Conference on Object-Oriented Programming (ECOOP 2024).
Editors: Jonathan Aldrich and Guido Salvaneschi; Article No. 46; pp. 46:1–46:26



To make it feasible to mitigate timing channels, one common practice is to identify and rule out *dangerous code patterns* that lead to timing channels. Notably in cryptographic systems, a common countermeasure against timing attacks is to follow *constant-time disciplines* [1, 11], which rules out (1) branching on secret-dependent data, as well as (2) accessing memory with secret-dependent offset (e.g., an array access with a secret-dependent index). For example, a secret-dependent branch `if s[i] then x = 2` may be identified by noting that `s[i]` is the i -th bit of the private key, hence violating the first constant-time discipline. We might further replace it with `x = s[i]*(2-x)+x`, or `x = (-s[i]&2)|((s[i]-1)&x)`, which are both functionally equivalent to the original code. While the violation in this example is easy to spot as it directly uses the secret value s , detecting constant-time violations and rewriting them in a secure way in general is still error-prone, as evidenced by timing leak in manually validated code [27] as well as a sequence of timing-related patches where early patches introduce new vulnerabilities that are fixed by later ones [36]. Hence, more rigorous and automated techniques are necessary for security.

Motivated by the need for rigorous and automated tools, designing and developing automated tools for detecting and sometimes repairing constant-time violations has been an active research area. Observing the relation between constant-time disciplines and non-interference [16], most automated tools rely on some form of information flow analysis to detect constant-time violations (i.e., to detect “tainted” branch conditions and memory addresses). For example, ct-verif [1] uses a sound and complete reduction on the source code to verify constant-time disciplines as a safety problem, via off-the-shelf verification tools. FACT [11] is a domain-specific language that uses a static information flow type system to detect constant-time violations. Constantine [7] deploys a dynamic taint analysis [7], while FlowTracker [32] uses an optimized program dependence graph (PDG) to identify constant-time violations. If we narrow down the scope further and focus on *cache-based* timing channels only, it is also common to identify constant-time violations first, before further refining the results to those that are vulnerable to cache-based attacks only. For instance, SC-Eliminator [46] uses a static taint analysis to identify “leaky conditional statements” (i.e., sensitive branches) and “leaky lookup-table accesses” (i.e., sensitive memory addresses via array accesses) before applying a (more costly) static cache analysis on leaky lookup-table accesses to filter out the ones that are not vulnerable to cache-based timing attacks (though they might still be vulnerable to other variants of timing attacks; see Section 2.3). The same applies to SpecSafe [9] and oo7 [43], which used static and dynamic taint analysis respectively before applying more costly cache analysis to refine their results.

With ample tools that use information flow analysis to detect constant-time violations, the *precision issue of those tools* is still under-investigated to the best of our knowledge. Yet, to be practical, the precision issue is just as important as soundness, especially given the common wisdom that static information flow analysis usually comes with high false positive rates when being applied to real-world applications [33]. To be more concrete, the following research questions still linger when cryptographic library developers plan to develop or adopt an information flow analysis for the purpose of detecting constant-time violations:

- RQ1:** To analyze cryptographic libraries, what are the impacts of various design decisions (e.g., field-sensitivity, declassification, context-sensitivity) on the analysis precision?
- RQ2:** Which approach (e.g., logic-based formal verification, PDG-based analysis) should they follow in order to achieve better trade-offs between precision and performance?
- RQ3:** Is it possible to improve the precision of existing tools without sacrificing soundness and performance?

To address these research questions, we start from a baseline PDG-based static information flow analysis called PIDGIN [18], with minor extensions that use its resulting sensitivity of registers and memory blocks to detect violations of constant-time disciplines. With the baseline implementation, we thoroughly study the root causes of false positives produced by the baseline and improve it with various features such as field-sensitivity, declassification and flow-sensitivity to reduce its false positive rate. The precision study is performed on a benchmark consisting of various implementations of modular exponentiation used in popular cryptographic systems (Libgcrypt, OpenSSL, mbedTLS and BearSSL), which are known to be vulnerable to timing attacks, as well as a set of automatically repaired code from Constantine [7], which are expected to be constant-time. The precision study shows that field-sensitivity and declassification are the most effective features. As a result from the precision study, we built CtChecker, a precise, sound and efficient information flow analysis for constant-time programming. CtChecker was implemented based on PIDGIN, with several precision improving features introduced. CtChecker targets LLVM intermediate representation (IR), which allows it to analyze various source code languages with compatible compiler front-ends. In summary, CtChecker reduces the false positives of the baseline by 67.6%, and we observed that the remaining ones are mainly due to imprecision in the sound points-to analysis used by CtChecker. Moreover, CtChecker detects true positives in 6 repaired programs produced by Constantine, which was not revealed before to the best of our knowledge.

A1: Field-sensitivity and declassification are the most effective features that can improve analysis precision, while the precision of point-to analysis (used by sound information flow analysis) has a considerable impact on the overall precision.

To study the remaining two research questions, we first compare the precision of CtChecker with respect to ct-verif [1], which first transforms source code to target code via a sound and complete reduction, and then verifies safety properties on the target code via off-the-shelf verification tools. Despite the fact that the reduction is sound and complete, and the common wisdom that logic-based formal verification generally offers superior precision than static program analyses that are built on approximation of program semantics, CtChecker turns out to be both more precise and more efficient compared with ct-verif [1]. Digging into the results, we found that ct-verif uses *loop invariant heuristics* to verify the code after reduction, which introduced higher false positive rates than CtChecker. Moreover, PDG-based approach is more appealing for detecting constant-time violations as it can pinpoint those violations in the code, while verifiers only produce a binary result of the existence or absence of violations.

A2: Based on our empirical study on state-of-the-art tools, we found that PDG-based information flow analysis offers the best trade-off between precision and performance compared with logic-based approach.

We further compare CtChecker with CacheS [44] and SC-Eliminator [46] on the benchmarks that those tools were evaluated on. Both tools are *soundy* as CacheS is built on lightweight but unsound memory model, and SC-Eliminator assumes no information flow via aliasing. Though CtChecker is built on a sound points-to analysis, we found that CtChecker reports very similar positives with those reported by CacheS [44], even though CacheS is built on complex abstraction interpretation with an unsound memory model. Compared with the static taint analysis used by SC-Eliminator, CtChecker reports fewer positives in 6 programs and the same positives in 9 programs in the SC-Eliminator benchmark. CtChecker reports more positives in 2 programs, but they are all true positives missed by SC-Eliminator, since it does not propagate information flows via aliasing.

A3: CtChecker improves the precision of existing tools without sacrificing soundness and performance. Compared with CtChecker, existing solutions either fall short on the high false positive rates with sound static method [1], or inherit the unsoundness of dynamic

method [45, 37], or simplified but unsound memory model [44, 46]. Notably, the precision of CtChecker is close to those built on unsound memory models, even though they have the advantage of possibly reporting fewer false positives by sacrificing soundness.

In summary, this paper makes the following contributions:

- Design and implementation of CtChecker, a precise, sound and efficient static information flow analysis for constant-time programming. The source code is made publicly available¹.
- Identification of the imprecision sources (e.g., field-sensitivity, declassification and flow-sensitivity) of constant-time analysis on cryptographic libraries (Section 3), and improvement of overall analysis precision based on the findings. Overall, field-sensitivity and declassification are the most effective features that improve precision, while combining multiple features enables CtChecker to reduce the total false positives compared to the baseline by 67.6% (Section 4).
- Comparison between CtChecker and state-of-the-art tools with similar goals. Evaluation results suggest that PDG-based information flow analysis offers the best trade-off between precision and performance compared with logic-based approach (e.g., ct-verif [1]) and abstract interpretation (e.g., CacheS [44]). Moreover, its precision is close to tools that are built on unsound memory models (Section 4).
- Evaluation of CtChecker on automatically repaired code from Constantine [7] reveals new timing channels that are not reported before (Section 4).

2 Background

2.1 Information Flow Analysis

Information flow analysis tracks interactions of information throughout a program. Given the confidentiality of program inputs, an information flow analysis tracks which data have been computed from confidential information or its derivatives. In general, information flow analysis handles complex confidentiality and/or integrity policies which can be formalized as a security lattice [13], while its simplest setting, known as *taint analysis*, only handles a two-level confidentiality lattice with “public” and “secret” labels only. A *sound* information flow analysis typically enforces *non-interference* [16] or its variants. Intuitively, non-interference guarantees that information labeled with higher confidentiality (or lower integrity) has no influence on information labeled with lower confidentiality (or higher integrity).

Listing 1 Example of Explicit and Implicit Information Flow.

```
// key = sensitive information
x = key + 1;
y = 0;
if (key > 100) {
    y = 1;
}
```

There are two kinds of information flows: explicit and implicit, as illustrated in the example shown in Listing 1. In this simple example, `key` is the only confidential input. We can see that `x` is directly computed from `key`, forming an *explicit* information flow. On the other hand, the variable `y` is assigned to a public constant in the true branch. However, due to the fact that the assignment occurs only in a branch whose condition is dependent on `key`, an attacker with the ability to observe the value of `y` can infer if the value of `key` is above 100 or not. In this case, the confidential data *implicitly* flows into `y`.

¹ <https://github.com/psuplus/CtChecker>

A variety of information flow analyses have been implemented with different methodical approaches. The utilization of program dependence graph (PDG) to detect the flow of information inside a program can be found in works such as PIDGIN [18] and FlowTracker [32]. Another approach is to use reduction techniques such as self-composition [5, 1] and product programs [4, 48] to reduce the information flow problem to safety properties, which can be verified by formal methods. Type-based approach, another more traditional method, can be found in various implementations [23, 42, 30]. Besides the static sound approaches above, dynamic taint analysis tracks information flow [14, 24, 39] at program execution time.

In this paper, we focus on the PDG-based approach and logic-based approach as they have been used in existing *automated and sound* tools that detects timing channels [1, 32]. Some prior work uses type-based approach to detect timing channels in software [49] and hardware [51] designs, but the precision of type systems is usually limited [33, 20]. They require type annotations from programmers, which is time-consuming, and pinpointing the root cause of type errors is a nontrivial task [50, 25, 34].

2.2 Timing Channels in Cryptosystems

A timing channel is a side channel in which an attacker uses program execution time to learn information about sensitive data. In some implementations of sliding window exponentiation, for example, the sequences of squares and multiplies can be measured due to differences in each method's execution time. This attack can be illustrated in the source code from an old OpenSSL implementation of sliding window exponentiation, shown in Listing 2.

Listing 2 Square and Multiply Timing Channel.

```

1  for (i = 1; i < bits; i++) {
2      if (!BN_sqr(v, v, ctx))
3          goto err;
4      if (BN_is_bit_set(p, i))
5          if (!BN_mul(rr, rr, v, ctx))
6              goto err;
7 }
```

The for-loop here iterates through each bit in the confidential exponent p . In each iteration, it first computes the square of v , and if the i -th bit of p is set, an additional multiplication computation is executed before proceeding to the next loop iteration. Since the extra multiplication computation is *only performed* when the i -th bit of p is set, the code is vulnerable to a timing attack which utilizes the “side effects” of the extra computation on timing to rebuild the entire private key (e.g., [6]).

Another common kind of timing channels in cryptography algorithm implementations root from array accesses being indexed with an offset derived from secret. The reason is that when accessing the memory, different indices may cause the loading or eviction of data into or from different cache lines in the data cache. By observing such behaviors, an attacker could reveal secret data; this variant of timing attacks is also known as *cache attacks*. The code snippet below is an example which is vulnerable to cache attack, where a public array $Sbox$ is accessed with secret key RK :

```
RK[4] = RK[0] ^ Sbox[(RK[3] >> 8) & 0xFF];
```

As different values of $RK[3]$ can lead to different cache lines to be fetched, a cache-probing attack can be launched to learn the value of $RK[3]$. The code snippet represents real timing vulnerabilities in encryption algorithms such as AES [17].

2.3 Constant-Time Disciplines and Cache-Specific Analysis

As timing channels are revealed by the execution time of a program, and many hardware features (e.g., data/instruction cache, CPU pipeline and cache bank) can affect timing, manually reasoning about timing channel vulnerabilities is extremely challenging, if possible at all. So as a practical countermeasure against timing channels, the threat model of constant-time discipline [1, 11] defines two constant-time violations of programs which can be exploited by attackers: (1) secret-dependent branch conditions, and (2) secret-dependent memory addresses. The constant-time disciplines are widely adopted in security-critical cryptographic systems [1, 11] due to a few benefits:

- It is *agnostic* to hardware configurations (e.g., cache configuration and replacement policy and features (e.g., cache, pipeline and cache bank) that are utilized by timing attacks).
- It provides a security abstraction that is more attractable both for programmers and for rigorous program analysis. For example, it is a common practice to use information flow analysis [1, 7, 11, 32] to detect constant-time discipline violations: it is sufficient to tag branches and memory accesses that use tainted values.

We note that under a *weaker threat model* that focuses on cache-based timing attacks only, various cache analyses [8, 35, 46] has been developed to detect if confidential information has impact on the cache hit/miss behaviors. For instance, SC-Eliminator [46] assumes a weaker threat model where an attacker only observes the number/type of instructions and cache hits/misses. Consider `preload A; A[secret]`, where `preload A` loads the whole array `A` into the cache. The code is secure per SC-Eliminator’s threat model since `A[secret]` always hits the cache. But it is insecure against other sophisticated attacks, such as the CacheBleed attack [47] that exploits cache-bank conflicts, and traffic analysis on the memory bus.

Despite the differences in their threat models, we note that cache analysis sometimes performs a static taint analysis first to identify array indices that are potentially tainted by sensitive data. Then, with the results from the taint analysis, a more costly cache analysis is performed to determine cache hits/misses. A positive from the taint analysis is removed if the cache analysis decides that this positive is a cache must-hit [35, 46]. Hence, precision improvements in sound and efficient static taint analysis, the focus of this paper, could also improve the precision of existing cache analysis, despite their very different threat models.

3 CtChecker Design

In this section, we depict the design details of CtChecker. We first describe the general workflow of CtChecker. Then we highlight the unique challenges and their solutions, and finally discuss the precision of CtChecker.

3.1 General Workflow

CtChecker first captures information flows by identifying the information flow introduced by each instruction according to its semantics. The captured flows are represented as a set of constraints, where each constraint element represents the sensitivity of registers or memory blocks. A least solution that satisfies all constraints is then computed. Lastly, all branch conditions and memory accesses are checked against the least solution to see if any sensitive information was used in them, resulting in violations of constant-time discipline.

```
<pointer> = getelementptr inbounds <ty>, ptr
    <ptrval>{}, [inrange] <ty> <idx>}*
```

Flow: $V(\text{ptrval}) \rightarrow V(\text{pointer})$

```
store <ty> <value>, ptr <pointer>
```

Flow: $V(\text{value}) \rightarrow D(\text{pointer})$

```
<result> = load <ty>, ptr <pointer>
```

Flow: $V(\text{pointer}) \cup D(\text{pointer}) \rightarrow V(\text{result})$

■ **Figure 1** Memory-Related Specification for LLVM IR.

3.1.1 Capturing Information Flows

The information flow specification of each instruction describes the sources (i.e., where information come from) and the sinks (i.e., where information go to). Based on the semantics of each instruction, the sources and sinks are usually easy to identify: instruction operands being read are sources and operands that are written to are sinks. However, since pointers can point to different data (in the memory) and the data to which they point to can change, a concrete information flow specification needs to distinguish three categories of elements, namely 1) the operand itself (e.g., a register), 2) the memory location that a pointer directly points to, and 3) all memory locations that are reachable from a pointer (through pointer arithmetics). Hence, we define three functions (V, D, R) which return the elements to be constrained for a value, respective to the three categories.

- $V(x)$, the value associated with the operand x .
- $D(p)$, the memory block that a pointer p points to.
- $R(p)$, the set of all reachable (i.e., accessible) memory blocks from a pointer p .

In order to correctly compute D and R , a sound points-to analysis is employed. Formally, the points-to analysis creates a directed graph $G = (P, M, E)$ from the code being analyzed, where node set P represents all pointer values, node set M represents all memory blocks being allocated, and edge set E links nodes in P to nodes in M (i.e., the points-to relation), as well as links connecting memory blocks, as a memory block can also hold a pointer.

To compute the function $D(p)$, we locate $p \in P$ and return the set $\{m \mid m \in M \wedge (p, m) \in E\}$. Similarly, to compute the function $R(p)$, we return the set of memory blocks that are *reachable* from p in G . For example, assume a pointer p_1 pointing to a pointer field of a data structure, $D(p_1)$ will be the memory block storing the pointer only, while $R(p_1)$ contains all the memory blocks that can be recursively reached by p_1 through node traversal, which include both the memory block storing the pointer and the data that the pointer points to.

For most IR instructions, information flow is captured by a simple value flow, $V(\text{source}) \rightarrow V(\text{sink})$. For memory-related instructions (e.g., the `getelementptr` (GEP), `store` and `load` instructions in LLVM IR), the more complicated specification is given in Figure 1. Specifically, for GEP instruction that calculates the memory address of a subelement in an aggregate data structure (i.e., array and structure) from the base pointer and the index of the target subelement, the sink $V(\text{pointer})$ is the address directly pointed to by p . For `store` and `load`, function D is used to specify the source and destination of the memory write and read respectively. Function R is used for function calls, which we discuss in Section 3.2.4.

3.1.2 Constraints and Their Least Solution

With the information flow specification, CtChecker creates a set of *constraints* from sources to sinks for each instruction. For each register and memory block $m \in M$, a distinguished *constraint element* is created (i.e., a one-to-one mapping). For an information flow $source \rightarrow sink$, a constraint is generated as $E_{source} \leq E_{sink}$ where E_{source} and E_{sink} are the corresponding constraint element of the *source* and *sink* respectively. The \leq symbol represents a partial ordering on the constraint elements and each constraint element can either be L (public) or H (secret), where $L < H$ and $H \not\leq L$. Additionally, CtChecker identifies the initial sensitive data m (e.g., secret keys) and generates constraints of the form $H \leq E_m$. For example, code snippet “`k=secret; x=k; y=0;`” will generate three constraints $H \leq E_k$, $E_k \leq E_x$ and $0(\text{constant}) \leq E_y$. These generated constraints are then added to the set of constraints, which are used to find the least solution.

The least solution, if exists, can be computed with linear-time algorithms, such as the Rehof-Mogensen algorithm [31]. By definition, the *least solution* provides the *least* confidentiality level of each constraint element, where the set of constraint elements with the least level H are considered to contain sensitive information. For the code snippet above, the least confidentiality level of `k`, `x` and `y` are H , H , and L , respectively. As a result, `k` and `x` are considered containing sensitive information.

3.1.3 Checking the Constant-Time Discipline

With the least solution at hand, CtChecker can check whether the analyzed program adheres to the constant-time discipline by examining all branch conditions and memory accesses, where a violation is found if a branch condition has level H , or a memory address being loaded from or stored to has level H . CtChecker reports all locations in terms of the line number in the source code regarding violations of the constant-time discipline.

3.2 Challenges and Solutions

The general workflow above presents the foundation of a *sound* analysis of constant-time programs. However, to develop an *useful* analysis, we need to address the following challenges.

3.2.1 Field-Sensitivity

In the naive analysis above, function D is modeled as $D(\text{pointer})$, the aggregated data structure that `pointer` points to. However, it is common for cryptographic libraries to store secret and public information in the same structure. In Listing 3, consider the `gcry_mpi` struct of Libgcrypt, where the array under `expo->d` in the modular exponentiation function holds secret value, while all other metadata fields are public. The naive approach would constrain `expo` as a single entity, reporting both branches at lines 8 and 9 as secret-dependent branches. However, the branch on `expo->flags` is not secret, resulting in a false positive.

To address the issue, both the sound points-to analysis and information flow analysis need to be *field-sensitive*, meaning that they both need to differentiate different fields in a structure. In particular, points-to analysis creates one separate memory block for each *subelement* in aggregate data structures, and correspondingly, CtChecker creates one distinct constraint element for each memory block from points-to analysis. By retrieving the offset and size information when possible (using `idx` operand from the corresponding GEP instruction of a `store` or `load` instruction as well as type information), CtChecker refines function D as $D(\text{pointer}, \text{offset}, \text{size})$, providing the necessary lookup information into the memory nodes in G , constructed by a field-sensitive points-to analysis. Function R is refined in a similar way.

■ **Listing 3** Field-Sensitivity Issues.

```

1 struct gcry_mpi {
2     ...
3     unsigned int flags;
4     mpi_limb_t *d;           // secret value
5 };
6
7 struct gcry_mpi *expo;
8 if (expo->flags) ... // not secret-dependent
9 if (expo->d[i]) ... // secret-dependent
10 if (expo->d) ... // not secret-dependent

```

One remaining subtlety is to distinguish pointer values and the memory blocks that they point to. For example, it is common to store private keys at the public memory address (i.e., revealing the addresses of the keys does not reveal their values). To prevent CtChecker from over-tainting these addresses (e.g., line 10 in Listing 3), CtChecker follows a two-phase approach. In the first phase, CtChecker propagates the addresses that are potentially storing sensitive information. In the second phase, when the data in these addresses is accessed, the tainted information will then be tracked from these accesses.

We note that while field-sensitivity can be enabled on many cases, a sound analysis inevitably needs to sacrifice field-sensitivity in cases where type information is missing, or aliases have inconsistent types, for instance. In these cases, refined function $D(\text{pointer}, \text{offset}, \text{size})$ resorts to its basic version without `offset` and `size` (i.e., D retrieves all fields in the structure that `pointer` points to).

3.2.2 Declassification

In real-world applications, strict information flow analysis should be relaxed to allow information flows that reveal limited or intended amount of sensitive information. This relaxation is known as *declassification*. CtChecker supports a *whitelisting* mechanism, where a user-provided whitelist (often provided by a programmer with domain knowledge) consists of variables that are derived from tainted data but are considered harmless. For all cryptographic libraries, we add variables storing *key sizes* but nothing else to the whitelist. These variables are manually checked to make sure that they do not contain key content themselves.

With a whitelist, CtChecker removes constraints that are associated with whitelisted variables after constraint generation. This way, not only the whitelisted variables are considered public, but also variables derived from them. The branches based on whitelisted variables or their derivatives are not reported by CtChecker.

3.2.3 Flow-Sensitivity

A variable might store both sensitive and public values at different program points, leading to imprecision issues. Consider the code snippet where both branches are dependent on `i`.

```

1 int i = key;
2 if (i == 0) ... // secret-dependent
3 i = 10;
4 if (i == 0) ... // not secret-dependent

```

A naive information flow analysis generates the constraints $H \leq E_i$ and $L \leq E_i$, where `i` throughout the program shares the same constraint element E_i . The least solution of the constraints is $E_i = H$, meaning that `i` is potentially sensitive. As a consequence, both branch conditions at lines 2 and 4 are marked as violations of constant-time discipline.

```
declare <RetType> @<FnName> ([arg list])
```

Flow: $V_{args} \cup R_{args} \rightarrow T_{ret} \cup R_{args}$

$$V_{args} = \bigcup_{i=0}^N V(arg_i)$$

$$R_{args} = \bigcup_{i=0}^N \{R(arg_i) : type(arg_i) = \text{pointer}\}$$

$$T_{ret} = \begin{cases} V(\text{retval}) & \text{RetType = primitive} \\ R(\text{retptr}) & \text{RetType = pointer} \end{cases}$$

 **Figure 2** Unknown Function Specification for LLVM IR.

Observing that the root cause of the issue above is lacking flow-sensitivity, CtChecker utilizes existing compiler support to improve flow-sensitivity. In LLVM, all registers are in static single-assignment (SSA) form. Hence, CtChecker can utilize LLVM’s `mem2reg` pass, which transforms the IR code by turning the standard `alloca-store-load` instruction sequences on memory (e.g., assignment and use of `i` in the example above) into simple register assignments. With the code transformation, the two `i`’s are named as `i.0` and `i.1`, respectively. Hence, two constraints are generated: $H \leq E_{i.0}$, $L \leq E_{i.1}$, and the least solution is $E_{i.0} = H, E_{i.1} = L$. Therefore, only the branch at line 2 which uses `i.0` is marked.

However, we note that due to aliasing and other subtleties in C language’s memory model, enabling flow-sensitivity on pointer-based accesses while maintaining soundness is much more challenging, which is beyond the scope of this paper.

3.2.4 Unknown Functions

The analyzed code often calls to external functions whose source code is either unavailable, or not covered by the analysis. To soundly capture information flows with the absence of some function definitions, we need to constrain possible flows in those missing functions. Obviously, input arguments can flow to return values. Moreover, if an argument or the return value is a pointer, any value that is *reachable* from the pointer-argument might flow to all *reachable* values from the pointer-return (e.g., through pointer arithmetics and memory writes). Hence, the information flows with absent function implementation is specified as the rule in Figure 2, where reachable memory from pointer-arguments are both sources and sinks of information flow, while reachable memory from pointer-return are sinks.

3.3 Precision of CtChecker

While CtChecker is empowered by various techniques above to improve its precision while maintaining soundness, false positives are still unavoidable like any nontrivial static program analysis.

One potential source is from a sound points-to analysis. To be sound, the points-to analysis must mark memory nodes as collapsed when the type information is inconsistent or missing. Once a node is collapsed in the points-to analysis, field-sensitivity is lost on the memory node. Moreover, whenever the points-to analysis fails to distinguish two different pointers’ corresponding memory nodes, it merges them into one node, resulting in an over-approximation of the taint analysis.

Second, CtChecker is a context-sensitive interprocedural analysis. However, when a callee function is invoked multiple times within the same caller function with different arguments, CtChecker only creates one context for all calls. This leads to the same indexed arguments of the multiple calls being aliased in the points-to analysis. As a result, a similar over-approximation is observed.

Furthermore, the adoption of LLVM’s `mem2reg` pass only enables flow-sensitivity on non-aggregate type memory. Memory accesses involving GEPs are still flow-insensitive.

While the above limitation prevents us from removing all false positives, we observe that CtChecker is able to outperform existing sound analysis. We provide the evaluation details next in Section 4.

4 Evaluation

4.1 Implementation

We implement CtChecker on PIDGIN [18], a static information flow analysis that integrates a query language into program dependence graphs (PDGs). However, PIDGIN lacks the precision-enhancing features discussed in Section 3.2, which we implement with an additional 2100 LOC in C++.

One implementation choice of static information flow analysis is the points-to analysis to derive sound approximation of memory blocks that a pointer might point to. The two mainstream points-to analysis algorithms are the unification-based Steensgaard’s algorithm [38] and the inclusion-based Andersen’s algorithm [2]. Both algorithms have implementations that are both context- and field-sensitive in order to gain better precision. For example, DSA [22] is a field- and context-sensitive points-to analysis based on Steensgaard’s algorithm with heap cloning. On the other hand, Andersen’s algorithm is generally considered more precise but also costly. Tools such as SVF [40] provide precision from context- and field-sensitivity without sacrificing much performance. Noting that DSA is used in our baseline PIDGIN and ct-verif [1], a representative logic-based tool, CtChecker is also built on top of DSA to make fair comparison with them (see the comparison with PIDGIN in Section 4.4 and ct-verif in Section 4.5). We leave the study of the impact of points-to analysis on constant-time analysis as future work.

4.2 Benchmarks

We evaluated CtChecker on two sets of benchmark programs. The first benchmark set consists of code taken from four cryptographic libraries, i.e., BearSSL v0.6 [29], Libgcrypt v1.10.1 [12], mbedTLS v3.2.1 [41] and OpenSSL v1.1.1q [15]. Among the four libraries, Libgcrypt and OpenSSL have widespread use, mbedTLS is built for embedded platforms, and BearSSL is less popular but it claims to be a constant-time cryptographic library [28]. In particular, the code consists of the modular exponentiation implementations of each library, where 4 implementations are taken from OpenSSL, as it is the only library that has various implementations for the same functionality. In the benchmark, the exponents in the modular exponentiation computation is marked as confidential sources. According to the definition of constant-time discipline, the sinks of the analysis are simply branch conditions and memory addresses.

The second benchmark set consists of code generated by a constant-time rewriter Constantine [7], which automatically identifies timing channels in the source code and repair them to follow the constant-time discipline². To the best of our knowledge, this is the first work to analyze the rewritten code by constant-time rewriters. Due to an incompatible LLVM version used by Constantine [7], we used an off-the-shelf C-backend [19] to translate their rewritten IR back to C source code whenever possible.

4.3 Evaluation Setup

We answer four research questions with the evaluation:

- Q1: Impact of separate features.** How do field-sensitivity, declassification and flow-sensitivity affect the analysis precision as separate features? Will including additional source code improve the analysis precision? Does CtChecker improve the precision of its baseline (i.e., PIDGIN)?
- Q2: Comparison with state-of-the-art.** Does CtChecker improve the precision when compared with ct-verif [1], a sound verifier for constant-time programming? Does CtChecker produce comparable or even more precise results when compared with CacheS [44] and SC-Eliminator [46]), both are built on simplified but unsound memory model?
- Q3: Precision.** How many false positives does CtChecker produce? What are the origins of the remaining false positives?
- Q4: Scalability.** Does CtChecker scale to real-world codebase with moderate size?

4.4 Impacts of Analysis Features

To answer Q1, we first create three variants of CtChecker, where only one feature among field-sensitivity (FS), white-list (WL) and flow-sensitivity (FL) is enabled. One extra feature, which is external to CtChecker, is how much code does it cover in the analysis. To study the impact of code coverage, we create two versions of each library implementation: one only includes the essential code that is necessary to compile the modular exponentiation implementation, while the other version (SRC) includes utility functions such as the multi-precision integer (mpi) or big number (bn) libraries³. The evaluation results are summarized in Table 1.

The improvement for field-sensitivity alone (column FS) is smaller than expected: while all implementations allocate secret and non-secret values in same structures, only four implementations (Libgcrypt, mbedTLS, BearSSL and OpenSSL-MontConstTime) observe some improvements. The reason is largely due to the lack of utility function implementations: both points-to analysis and information flow analysis remain very conservative without callee’s implementation, making it hard to differentiate read/write effects on each individual data field.

All four libraries saw a considerable reduction in positives when whitelist was used (column WL). The removed positives only leak key sizes, which are explicitly declassified in the whitelist. OpenSSL and mbedTLS saw a slight reduction with flow-sensitivity enabled (column FL).

² We pick Constantine [7] instead of other available rewriters such as SC-Eliminator [46], Lif [35] as they both assume a weaker threat model that only tackles cache attacks (see Section 2.3). In particular, their rewritten code with prefetching technique still violates constant-time disciplines.

³ The only exception is on BearSSL, which remains the same for both versions for two reasons: (1) the modular exponentiation function only contains high-level code that makes up fewer than 20 lines of code. An analysis on it alone does not generate any meaningful result, and (2) BearSSL has a much smaller codebase compared with other libraries.

Table 1 Number of positives based on features. Each column represents the analysis result with some features enabled. *Base*: the baseline analysis, *FS*: field-sensitivity, *WL*: whitelist, *SRC*: adding extra source code, *FL*: flow-sensitivity, and *All*: all features enabled. *TP* represents the true positives and *Reduction* computes the reduction rate of false positives, i.e., $(\text{Base} - \text{All}) / (\text{Base} - \text{TP})$.

Library	Base	FS	WL	SRC	FL	All	TP	Reduction
Libgcrypt 1.10.1	66	46	55	76	66	30	6	60.0%
mbedTLS 3.2.1	50	45	48	33	45	10	4	87.0%
BearSSL 0.6	18	15	6	18	16	3	1	88.2%
OpenSSL 1.1.1q								
Reciprocal	14	14	3	20	13	9	2	41.7%
Mont	45	45	36	37	44	25	2	46.5%
MontConstTime	36	27	28	29	34	18	0	50.0%
MontWord	4	4	2	15	4	12	1	-267%
binsec/aes_big	0	0	—	—	0	0	0	—
binsec/des_tab	51	26	—	—	51	26	24	92.6%
binsec/tls-rempad-luk13	7	6	—	—	7	6	6	100%
appliedCryp/3way	41	0	—	—	41	0	0	100%
appliedCryp/des	72	62	—	—	72	62	62	100%
appliedCryp/loki91	75	72	—	—	75	72	56	15.8%
ghostrider/findmax	0	0	—	—	0	0	0	—
ghostrider/matmul	4	0	—	—	4	0	0	100%
libg/des	448	432	—	—	448	432	432	100%
pycrypto/ARC4	20	19	—	—	20	19	19	100%
Overall	951	813	178	228	940	724	615	67.6%

Including additional source code (column SRC) does not necessarily reduce false positives: doing so in fact increases positive numbers for Libgcrypt, OpenSSL-Reciprocal and OpenSSL-MontWord. This somewhat surprising degradation comes from the imprecision of the underlying points-to analysis. The points-to analysis, using heap cloning technique, will merge distinct nodes that are processed by a common function. The analysis also merges nodes that are found to be in the same equivalence class. In the case of Libgcrypt, nodes that were considered distinct in the baseline test, end up aliased to the same node in the full source benchmark.

CtChecker enables all features and analyzes more than the essential code (i.e., it also covers utility function implementations), whose result is shown under the column “All”. For most libraries, CtChecker delivers the most precise result, with the exceptions of OpenSSL-Reciprocal and OpenSSL-MontWord. By inspecting the differences, we concluded that the reason is also due to undesirable effects in the points-to analysis when additional code is being analyzed.

4.5 Comparison with ct-verif

ct-verif [1] is one of the first sound tools for verifying constant-time properties; it uses the self-composition technique [5] to convert the constant-time property into a classical program verification problem. One reason for comparing with ct-verif is that it is also built on top of the DSA [22] points-to analysis; hence, the comparison focuses on the advantages of each approach, rather than some engineering details, such as the difference in the points-to analysis, in their implementations.

Table 2 Comparison with ct-verif. “Full-SRC” corresponds to full source in Table 1. “No-Undefined-Function” is the version where all function calls without sources are removed. “ct-verif-Verified” stands for all positives in ct-verif are removed, while “CtChecker-Verified” stands for all positives in CtChecker are removed.

Library	ct-verif				CtChecker			
	Full SRC	No Undefined Function	ct-verif Verified	CtChecker Verified	Full SRC	No Undefined Function	ct-verif Verified	CtChecker Verified
Libgcrypt 1.10.1	–	20	0	10	30	6	0	0
mbedTLS 3.2.1	OOM	5	0	1	10	4	0	0
BearSSL 0.6	3	3	0	0	3	3	0	0
OpenSSL 1.1.1q								
Reciprocal	–	2	0	0	9	2	0	0
Mont.	–	4	0	2	25	2	0	0
Mont. Const. Time	–	2	0	3	18	0	0	0
Mont. Word	–	1	0	0	12	1	0	0

One challenge in the comparison is that as a verification tool, ct-verif only reports whether the input program is constant-time or not⁴. To find the exact lines that ct-verif deems constant-time violations, a line-by-line check on the source code is needed. Whenever ct-verif reports a positive, we log the current line, modify it with some constant-time code, and run ct-verif again. The same strategy is applied to function calls that lead the control flow to other functions.

Even though we carefully make the changes so that the information flow remains the same, there is still a chance that the information flow might be changed while rewriting the code that has constant-time violations. For a fair comparison, both tools are running on the same rewritten code.

4.5.1 Results

Both ct-verif and CtChecker are sound analyses and we did observe that both tools report all true positives. The differences are on false positives. To evaluate the false positive reported by each tool, we consider four variants of the cryptographic libraries that we evaluated in Section 4.3. The results are summarized in Table 2.

Full-SRC contains the full source code of the libraries, including mpi libraries. ct-verif was only able to analyze BearSSL in this setting (recall that BearSSL has the smallest codebase among all libraries). ct-verif fails with an out-of-memory error on mbedTLS. As for Libgcrypt and OpenSSL, full source introduces a huge amount of source code. Since we have to manually go through the source code line by line with ct-verif to find offending lines. It is a prohibitive amount of work to get all positives. On the contrary, CtChecker could get all four libraries’ results, where the result on BearSSL is the same as what ct-verif reports.

No-Undefined-Function corresponds to the minimal source in Table 1. The difference here is that a function call will be removed if it calls an undefined function. The reason is to accommodate the difference in how the tools treat excluded sources. ct-verif treats a return value as sensitive even if there is no tainted argument used in this function call. CtChecker taints all reachable memory in the presence of pointer-values (even if the pointer itself is

⁴ When verification fails, ct-verif does generate some error messages. However, it is hard to decipher those messages and link them to the source code.

not tainted, see Section 3.2). Removing undefined function calls allows a fair comparison between the two tools. We note that for all libraries, CtChecker is consistently at least as precise as ct-verif, where CtChecker reports fewer positives in 4 out of 7 libraries.

ct-verif-Verified was created from column *No-Undefined-Function* by removing all positives reported by ct-verif, resulting in fully verified code that is constant time. Any positive reported by CtChecker on this version is expected to be a false positive. That being said, CtChecker reported no positive when all positives are removed in ct-verif.

CtChecker-Verified was created similarly from column *No-Undefined-Function* by removing all positives that are reported by CtChecker. Each program is a piece of verified constant-time code. Hence, the positives reported by ct-verif on this version are false positives (we also manually confirmed). ct-verif reports 16 false positives in total on the constant-time code.

To understand the possible causes for these false positives reported by ct-verif, we analyzed its output IR code and results. One reason is that memory nodes within an array are marked public with a constant length by annotation in ct-verif. When a loop is encountered, memory could be accessed with a loop variable. Then, ct-verif fails to determine whether a piece of accessed memory is within the public area or not, because it cannot infer how many iterations at most the loop will be executed. Loop invariants could be automatically computed to verify the range of loop variables. However, the loop invariant generation in ct-verif, based on a simple heuristic, might fail to verify secure code. Another series of false positives originates from how ct-verif handles `memcpy`, for which it will show arbitrary behaviors. For example, in the code snippet below, the addresses of `s`, `p1`, `p2` and `p3` are set to public. The contents of `p1`, `p2` and `p3` are also public.

```

1 int test(int *s, int *p1, int *p2, int *p3) {
2     int a=32;
3     memcpy(p1, p2, a);
4     if (p3[0]) dummy++;
5 }
```

After calling `memcpy`, the content of `p3` is tainted and line 4 will be reported, even though `p3` is neither an argument nor the return of `memcpy`.

In summary, CtChecker exhibited a considerable improvement in precision over ct-verif, a result apparent in the difference between the number of positives reported by each tool in the last three columns. Moreover, as discussed above, CtChecker is more user-friendly as it reports all positives in one shot, whereas using ct-verif to find all positives in source code is cumbersome.

4.6 Comparison with CacheS

CacheS [44] uses abstract interpretation to verify constant-time property. Notably, CacheS is a “soundy” analysis where “the implementation is unsound for speeding up analysis and optimizing memory usage, due to its lightweight but unsound treatment of memory”, quoted from the same paper. Also, it operates on a platform independent IR called REIL IR, which is lifted from x86 assembly code. We compare CtChecker against CacheS to show how the memory model and IR code could affect analysis results, see Table 3.

For Libgcrypt, CacheS reports line 19 in Listing 4 (line 682 in *mpi-pow.c*), where `e0` is derived from `secret`. But CtChecker ignores this line. The reason is that in LLVM IR, this line is neither compiled into a branch nor accesses memory with sensitive index. However, in REIL IR, `cmove` instruction is lifted to a branch with a condition that is derived from the `secret`, the reason that CacheS reports it.

■ **Table 3** Comparison with CacheS (★: low-risk positives; †: extra positive reported by CacheS).

Library	File	CtChecker Positives	CacheS Positives
Libgcrypt 1.10.1	mpi-pow.c	440	440
		559★	749★
		617	617
		641	—
		667	—
		—	682†
		702★	—
mbedTLS 3.2.1	bignum.c	2124	2124
		2127	2127
		2173	2173
		2182★	2182★
BearSSL 0.6	i32_sub.c	36	N/A
OpenSSL 1.1.1q			
Reciprocal	bn_exp.c	242	N/A
Mont.		262	
Mont. Word		398	
		419	
		1240	

We also observe that CacheS ignores a few high-risk positives reported by CtChecker. In Libgcrypt, two high-risk positives are overlooked, namely, lines 7 and 13 in Listing 4 (lines 641 and 667 in *mpi-pow.c*). In LLVM IR, the branch conditions at these lines are derived from the secret value and they are inside a loop, which leads to multiple-bit leakage. Similar to the previous case, the difference between two tools’ analysis targets leads to discrepancies in results. CacheS observes a `bsr` instruction in the assembly code, which is a constant-time instruction on most architectures.

CacheS employs a lightweight but unsound memory model, which avoids one issue of CtChecker: 69% of false positives of CtChecker are introduced by DSA. However, as a trade-off, this advantage may lead to false negatives in detecting high-risk vulnerabilities, though we did not observe any false negatives from CacheS in our evaluation.

4.7 Comparison with SC-Eliminator’s Taint Analysis

As discussed in Section 2.3, SC-Eliminator [46] and CtChecker assume different threat models. As a consequence, SC-Eliminator performs a taint analysis that identifies violations of constant-time disciplines first, before the results are further analyzed by a cache analysis. Here, we compare CtChecker with SC-Eliminator’s taint analysis as they both share the same functionality. The comparison is also meaningful as precision improvements in the taint analysis could help cache-analysis tools like SC-Eliminator to rewrite less code, which improves the performance of the product rewritten code.

Listing 4 Code snippet from Libgcrypt (*mpi_pow.c*).

```

1 count_leading_zeros (c0, e);
2 e = (e << c0);
3 c -= c0;
4 j += c0;
5
6 e0 = (e >> (BITS_PER_MPI_LIMB - W));
7 if (c >= W)
8     c0 = 0;
9 ...
10 count_trailing_zeros (c0, e0);
11 e0 = (e0 >> c0) >> 1;
12
13 for (j += W - c0; j >= 0; j--)
14 {
15     base_u_size = 0;
16     for (k = 0; k < (1<< (W - 1)); k++)
17     {
18         ...
19         base_u_size |= ( precomp_size[k] & (OUL - (k == e0)) );
20     }
21     ...
22 }
```

We build SC-Eliminator from source code⁵ with LLVM 8.0.1. The comparison was based on the benchmarks used in [46], see Table 4. Before analyzing the results, we note two major differences between CtChecker and SC-Eliminator’s taint analysis:

1. While CtChecker is built on a sound points-to analysis, SC-Eliminator’s taint analysis *does not* use any points-to analysis. The result is that the latter might miss taints that are propagated via aliasing.
2. While CtChecker is implemented as an interprocedural analysis, SC-Eliminator’s taint analysis is implemented as an *intraprocedural* analysis. Although an intraprocedural analysis is inherently free of context-sensitivity issues that we discuss further in Section 4.8, it requires manual efforts to label sensitive function parameters⁶, which is both time-consuming and error-prone.

Despite the differences above, both favor SC-Eliminator’s taint analysis in terms of precision, CtChecker reports fewer positives in 6 benchmark programs, while the two tools report the same number of positives on 9 of the benchmark programs. Surprisingly, SC-Eliminator reports less positives in two algorithm programs, namely, *anubis* and *cast5* in the Chronos library. The extra positives that CtChecker reports, as we investigate deeper, are true positives. *But due to the lack of a points-to analysis, SC-Eliminator missed them.* In other words, SC-Eliminator in fact has *false negatives*, which we elaborate next.

4.7.1 False Negatives in SC-Eliminator’s Taint Analysis

The lack of a points-to analysis sometimes breaks the propagation of information flow. The code snippet shown in Listing 5 contains a concrete true positive that is missed by SC-Eliminator. Here, the first parameter **key** of function **bar** is the tainted source. At line 8, the first element of **ctx->keys** is tainted by **key**. So, line 9 violates constant-time disciplines

⁵ <https://bitbucket.org/mengwu/timingsyn/src/master/>

⁶ To reduce the effort, with only a few hard-coded cases, SC-Eliminator’s taint analysis assumes that only the first parameter of every function to be tainted.

 **Table 4** Comparison with SC-Eliminator's Taint Analysis.

Library	SC-Eliminator		CtChecker
	Positives Reported	False Negatives	Positives Reported
appliedCryp/3way.c	4	0	3
appliedCryp/des.c	22	0	18
appliedCryp/loki91.c	8	0	7
chronos/aes.c	388	0	388
chronos/anubis.c	84	8	92
chronos/cast5.c	288	160	448
chronos/cast6.c	448	0	448
chronos/des.c	426	0	416
chronos/des3.c	400	0	390
chronos/fcrypt.c	128	0	128
chronos/khazad.c	40	0	40
libg/camellia.c	32	0	32
libg/des.c	144	0	144
libg/seed.c	8	0	8
libg/twofish.c	248	0	248
supercop/aes_core.c	412	0	409
supercop/cast-ssl.c	448	0	448

as the branch condition is tainted. However, SC-Eliminator misses the positive and leaves it unchanged in the rewritten code. Due to the lack of a points-to analysis, SC-Eliminator cannot infer that variable `keys` defined at line 7 and `ctx->keys` point to the same memory. This contrived example illustrates why SC-Eliminator misses those true positives in *anubis* and *cast5* in the Chronos library.

 **Listing 5** Example of a false negative in SC-Eliminator.

```

1 void do_something() {...}
2 typedef struct {
3     int **keys;
4     int n;
5 } CONTEXT_st;
6 void bar(int *key, CONTEXT_st *ctx){
7     int **keys = ctx->keys;
8     keys[0] = key;
9     if(ctx->keys[0][0] == 0)
10     do_something();
11 }
```

4.8 Analysis Precision

To answer Q3, we inspected each positive and categorized it into three kinds: low-risk, high-risk and false positive, where the first two are true positives, and their difference is in the severity of information leakage. In particular, a low-risk positive only reveals one bit of information while high-risk positives can leak multiple bits of secrets (e.g., a sensitive branch within a loop).

The classification result is shown in Table 5. CtChecker reports a total number of 724 positives, with 615 true positives and 109 false positives.

True Positives. CtChecker does find true positives in the rewritten code by Constantine. For example, line 5 in Listing 6 is a timing channel and line 17 is a cache side channel. At the beginning, `%idx` is an address calculated from `%sec`, which is derived from a secret

Table 5 Result Classifications. *Base* and *All* refer to the same columns in Table 1.

Library	Base	All	FP	Low-risk	High-risk
Libgcrypt 1.10.1	66	30	24	2	4
mbedTLS 3.2.1	50	10	6	1	3
BearSSL 0.6	18	3	2	0	1
OpenSSL 1.1.1q					
Reciprocal	14	9	7	0	2
Mont.	45	25	23	0	2
Mont. Const. Time	36	18	18	0	0
Mont. Word	4	12	11	0	1
binsec/aes_big	0	0	0	0	0
binsec/des_tab	51	26	2	24	0
binsec/tls-rempad-luk13	7	6	0	6	0
appliedCryp/3way	41	0	0	0	0
appliedCryp/des	72	62	0	62	0
appliedCryp/loki91	75	72	16	56	0
ghostrider/findmax	0	0	0	0	0
ghostrider/matmul	4	0	0	0	0
libg/des	448	432	0	432	0
pycrypto/ARC4	20	19	0	19	0

key. It is then cast to an integer type and is masked by 63. The result `%and` is tainted from the masking operation. At line 4, `%cmp`, the branch condition, is computed from `%and`, making the branch secret dependent. What makes the case more interesting is that the branch at line 5 *does not* exist in the original source code. Constantine adds the branch to check if `%and` satisfies certain property. If not, the execution will stop. For this reason, even though this branch is added into the main processing loop, it should be considered a low-risk positive. Later, `%and` is used to compute another address `%addptr` at line 11. Then, `%addptr` is accessed by `load` if the execution path comes from `forbody.pre`. This memory access is sensitive because the index is derived from secret even after the masking operation. All positives found in rewritten code follow the similar pattern.

Listing 6 Rewritten IR by Constantine from pycrypto/ARC4.

```

1 %idx = gep @stream_state, 0, %sec
2 %2 = ptrtoint %idx to i64
3 %and = and %2, 63
4 %cmp = icmp slt %and, and (sub (add (ptrtoint (@stream_state to i64),
319), ptrtoint (@stream_state to i64)), -64)
5 br %cmp, label %forbody.pre, label %exit
6
7 exit:
8 ...
9
10 forbody.pre:
11 %addptr = gep @stream_state, 0, %and
12 br label %forbody.body
13
14 forbody:
15 ...
16 %_ptr = phi [ %addptr1, %forbody ], [ %addptr, %forbody.pre ]
17 %3 = load volatile %_ptr

```

False Positives. While the overall false positive rate of CtChecker on all benchmarks appears low, we further studied the root cause of false positives in the cryptographic library benchmarks, which witness a higher rate of false positives, and found that the majority (63 of them) are caused by the imprecision of DSA, 26 by context-insensitivity, and 2 by

flow-insensitivity (see Section 3.3 for the major reasons of imprecision). Arguably, 69% of false positives due to DSA are unavoidable while we aim for a sound and scalable analysis. For example, when creating the data structure graph for the reciprocal method in OpenSSL, the structure representing the exponent p is collapsed. Without the field information, CtChecker have to conservatively taint the whole structure. Consequently, line 1 in Listing 7 (which corresponds to line 171 in `bn_exp.c`) is reported even though it only depends on `flag` field, not the data field.

Listing 7 False positives caused by collapsed memory and callsite-insensitivity in OpenSSL.

```

1 if (BN_get_flags(p, BN_FLG_CONSTTIME) != 0
2   || BN_get_flags(a, BN_FLG_CONSTTIME) != 0
3   || BN_get_flags(m, BN_FLG_CONSTTIME) != 0) {
4   BNerr(BN_F_BN_MOD_EXP_RECV, ERR_R_SHOULD_NOT_HAVE_BEEN_CALLED);
5   return 0;
6 }
```

29% of false positives are produced when a callee function is invoked multiple times within the same caller function with different arguments. In this case, DSA provides no callsite distinction. In Listing 7, `BN_get_flags` is called multiple times with p , a and m as the first parameter, respectively. Since p is tainted, lines 2 and 3 (lines 172 and 173 in `bn_exp.c`) are also reported. One fix is to distinguish all calling contexts in the static analysis, or inlining all function calls before the source code is being analyzed. However, both approaches will hurt the scalability of static analysis on large programs.

Other implementations of points-to analyses may be used to improve precision further in two different aspects: (1) to reduce the number of collapsed memory nodes, and (2) to provide finer-grained context-sensitivity on callsites. As an example, these improvements might remove false positives mentioned in the example of reciprocal method in OpenSSL above. However, as developing a more precise points-to analysis is beyond the scope of this paper, we leave it as future work.

The remaining 2 false positives are due to the lack of flow-sensitivity. As discussed in Section 3.2, LLVM's `mem2reg` pass only provides flow-sensitivity to some extent. For example, the tainted data in mbedTLS is the `d` field of variable `E`. Hence, line 1 in Listing 8 (line 1988 in `bignum.c`) is not key-dependent as it only returns the `s` field of E .

Listing 8 False positives caused by flow-insensitivity in mbedTLS.

```

1 if( mbedtls_mpi_cmp_int( E, 0 ) < 0 )
2   return( MBEDTLS_ERR_MPI_BAD_INPUT_DATA );
3 ...
4 while( 1 ) // Main processing loop
5 {
6   ...
7   MBEDTLS_MPI_CHK( mpi_select( &WW, W, (size_t) 1 << wsize, wbits ) );
8   ...
9 }
```

However, line 1 is falsely reported by CtChecker. Inside `mpi_select`, `s` field of `WW` is changed to the secret derived from `wbits` that is tainted. Since DSA treats `E` and `WW` as potential aliases, `s` field of `E` is also tainted. However, CtChecker is unable to distinguish *when* `s` field of `E` is tainted. Hence, it conservatively reports line 1 as a positive.

4.9 Scalability

To answer Q4, we evaluate the scalability of CtChecker on benchmark programs from Table 1 on a PC equipped with 2.30GHz Intel Core i7-11800H and 16 GB memory.

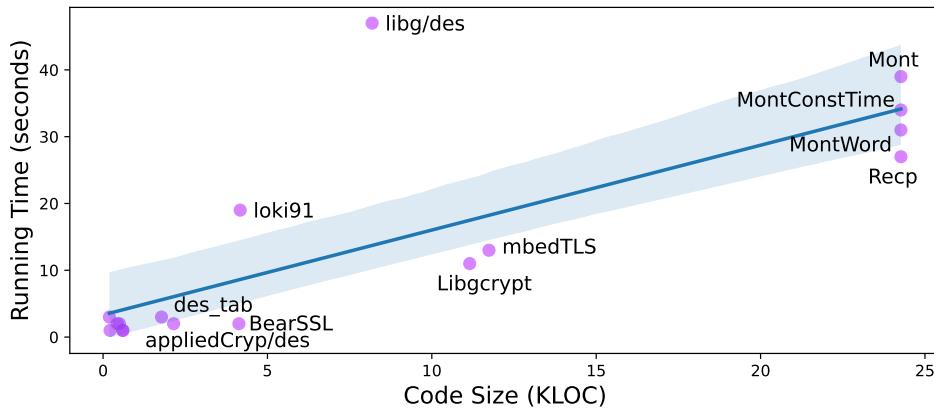


Figure 3 Running Time vs. Code Size of Benchmarks. Unlabeled data points are from Constantine rewritten code with smaller size.

For each cryptographic library, the experiment was done with both the full source (i.e., the SRC version) and the minimal source (i.e., the modular exponentiation code only). Among the tested benchmarks, BearSSL has the smallest codebase of about 4 KLOC, whereas the largest codebase is OpenSSL with around 24 KLOC for its full source version. The running time for these two libraries are 2 and 39 seconds, respectively. The comparison shows that processing time is small even for larger libraries. The libg/des code rewritten by Constantine, which has around 8.6 KLOC, had the longest running time of 48 seconds. Overall, the running time against code size of all benchmark programs being analyzed shows a close to linear trend, as shown in Figure 3.

In terms of memory consumption, the peak usage is around 150 MB when OpenSSL is being analyzed, during the constraint solving step. The result shows that CtChecker is scalable in both the spatial and temporal dimensions.

4.9.1 Running Time of Other Tools

We did not compare the running time and memory usage of CtChecker quantitatively with other tools since the underlying techniques are very different, making a direct comparison unfair. However, we discuss other tools' running time and memory consumption below.

ct-verif: For moderate-sized codebase like mbedTLS, the full source code makes ct-verif run out-of-memory (OOM) after several hours. Moreover, ct-verif stops execution once the first violation is found, making it hard to gauge its execution time if it were reporting all positives in one shot.

CacheS: As reported in [44], the running time for CacheS is at least 33.2 seconds when there is only one function being analyzed, up to 179.2 seconds if the execution runs successfully without timeout, and more than 5 hours if timeout occurs. The memory consumption is also significantly larger than CtChecker, where at least 620 MB of space was used. That said, we note that their reported numbers are collected from different experimental settings than ours, e.g., physical machine, analyzed code base, etc. Therefore, the performance numbers are not directly comparable.

SC-Eliminator: Its taint analysis is the fastest among all tools, which takes around 1 second for each of their benchmarks. The reasons are two-fold however. On the one hand, it does not utilize a points-to analysis, which sacrifices its soundness as we mentioned in Section 4.7.

On the other hand, it simply propagates taints when each instruction is analyzed, which leads to soundness issues in corner cases. For instance, given `x:=y; y:=secret` in a loop, SC-Eliminator fails to taint `x`, as the taint on `y` is discovered later in the analysis. CtChecker is built on more rigorous information flow analysis with constraint generation and solving. The points-to analysis and constraint generation/solving collectively consume most of the time for CtChecker.

Although a fair comparison on efficiency is infeasible, it is safe to conclude that CtChecker’s efficiency is better or at least comparable to other tools employing similar sub-components. SC-Eliminator is more efficient than ours and other competitors with a loss of soundness, as discussed above.

5 Related Work

5.1 Detecting Timing Side Channels

Both static and dynamic approaches are widely adopted to detect constant time violations. VirtualCert [3] and FlowTracker [32] are static tools built with formal methods. VirtualCert is flow-insensitive and is specially used for virtualized systems. FlowTracker focuses on optimizing the representation of implicit flows and is flow-sensitive. Almeida et al. [1] propose ct-verif, a static tool that employs self-composition for verifying constant-time property. It either accepts or rejects programs being verified, but it does not pinpoint the source code where the violations occur. Both VirtualCert and ct-verif require additional annotations to work. SecVerilog [51] is a language-based approach for checking hardware-level information flow violations. Somorovsky [37] presents a dynamic tool using fuzzing technique to detect implementation with constant-time violations. Dynamic methods could avoid false positives but are limited by their search space, which leads to unsoundness. Compared to these tools, CtChecker is a sound and generic non-constant-time code detection tool that does not require additional annotations.

5.2 Detecting Cache Side Channels

Cache-based side channels are another type of covert channel that could leak sensitive information to unintended parties. CacheD [45] is a trace-based analysis that identifies cache-based timing channels using taint tracking and symbolic execution. However, symbolic execution might not have the full coverage of execution paths. Brotzman et al. [8] propose a cache-aware symbolic execution (CaSym) that works on LLVM IR. CaSym is able to cover all execution path by introducing a technique that could transform a program with loops to its loop-free version. Both works report the location of vulnerabilities to make it easier for developers to fix them. CacheS [44] is a static analysis that could detect timing channels and cache-based channels. A novel abstract domain called Secret-Augmented Symbolic domain (SAS) is proposed to track sensitive information with high precision while remaining efficient. However, the unsound memory model it uses may cause false negatives. For comparison, CtChecker employs a sound points-to analysis, which makes it both sound and efficient.

5.3 Mitigating Side Channels

Another line of work involves mitigating side channels after vulnerabilities are detected. Cauligi et al. [11] propose a C-like DSL called FaCT. Its compiler is claimed to be able to compile secret-sensitive source code into constant-time LLVM bitcode. However, FaCT

requires libraries to be rebuilt in this language, making it impractical for existing libraries and legacy systems. Wu et al. [46] propose SC-Eliminator, a program rewriter that can eliminate both timing- and cache-based side channels. A constant time select function is proposed for secret-dependent branches. Cache-side channels are removed by preloading all elements in a lookup table. Soares et al. [35] point out that SC-Eliminator introduces out-of-bound memory accesses when doing the transformation. They put forward another rewriter called lif that ensures memory safety at the same time. Preloading methods fail when an attacker could evict cache lines after preloading and before accessing the data. Constantine [7] adopts a radical full linearization design. It focuses on how to maintain efficiency under the radical design. CtChecker does not repair programs when vulnerabilities are found. However, it could help the rewriters to identify problematic code locations more precisely, hence reducing the overhead of mitigation. Moreover, as we demonstrated in the evaluation, it can also serve as an efficient verifier for the code generated by those program rewriters.

6 Conclusion and Future Work

In this work, we build CtChecker, a sound, precise and scalable static information flow analysis for constant-time programming. Compared with traditional information flow analysis, CtChecker is equipped with various features to improve analysis precision on cryptographic code. The features effectively reduce false positive rates while maintaining analysis soundness. By inspecting remaining false positives, we observed that the majority is due to imprecision in the sound points-to analysis that CtChecker is built on.

For future work, a fraction of remaining false positives is due to a callee function being invoked multiple times within the same caller function with different arguments. We plan to investigate how to identify the instances where inlining such code might improve precision, with the insight that heterogeneous arguments to the callee function are the root cause of the imprecision issue. The selective inlining strategy likely will strike a good balance between precision and performance.

References

- 1 José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *USENIX Security Symposium*, pages 53–70, 2016.
- 2 Lars Ole Andersen. Program analysis and specialization for the c programming language. *Ph.D. Thesis*, 1994.
- 3 Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS ’14, pages 1267–1279, New York, NY, USA, 2014. ACM. doi:10.1145/2660267.2660283.
- 4 Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In *International Symposium on Formal Methods*, pages 200–214. Springer, 2011.
- 5 Gilles Barthe, Pedro R D’argenio, and Tamara Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(6):1207–1252, 2011.
- 6 Daniel J Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. Sliding right into disaster: Left-to-right sliding windows leak. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 555–576. Springer, 2017.
- 7 Pietro Borrello, Daniele Cono D’Elia, Leonardo Querzoni, and Cristiano Giuffrida. Constantine: Automatic side-channel resistance using efficient control and data flow linearization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 715–733, 2021.

- 8 Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut Kandemir. Casym: Cache aware symbolic execution for side channel detection and mitigation. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 505–521. IEEE, 2019.
- 9 Robert Brotzman, Danfeng Zhang, Mahmut Taylan Kandemir, and Gang Tan. Specsafe: detecting cache side channels in a speculative world. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–28, 2021.
- 10 David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.
- 11 Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. Fact: a dsl for timing-sensitive computation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–189, 2019.
- 12 GnuPG community. Libgcrypt. <https://gnupg.org/software/libgcrypt/index.html>, 2022.
- 13 Dorothy E. Denning. A lattice model of secure information flow. *Communication of the ACM*, 19(5):236–243, 1976.
- 14 William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):1–29, 2014.
- 15 OpenSSL Software Foundation. Openssl: Cryptography and ssl/tls toolkit. <https://www.openssl.org/>, 2022.
- 16 Joseph A Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20. IEEE, 1982.
- 17 David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games—bringing access-based cache attacks on aes to practice. In *2011 IEEE Symposium on Security and Privacy*, pages 490–505. IEEE, 2011.
- 18 Andrew Johnson, Lucas Waye, Scott Moore, and Stephen Chong. Exploring and enforcing security guarantees via program dependence graphs. *ACM SIGPLAN Notices*, 50(6):291–302, 2015.
- 19 JuliaHubOSS. Llvm c backend. <https://github.com/JuliaHubOSS/llvm-cbe>, 2018.
- 20 Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. In *Information Systems Security: 4th International Conference, ICIS 2008, Hyderabad, India, December 16-20, 2008. Proceedings 4*, pages 56–70. Springer, 2008.
- 21 Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.
- 22 Chris Lattner, Andrew Lenhardt, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. *ACM SIGPLAN Notices*, 42(6):278–289, 2007.
- 23 Andrew C Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, 1999.
- 24 James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, volume 5, pages 3–4, 2005.
- 25 Zvonimir Pavlinovic, Tim King, and Thomas Wies. Finding minimum type error sources. *ACM SIGPLAN Notices*, 49(10):525–542, 2014.
- 26 Colin Percival. Cache missing for fun and profit, 2005.
- 27 Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. Make sure dsa signing exponentiations really are constant-time. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1639–1650, 2016.
- 28 Thomas Pornin. Constant-time in bearssl. <https://bearssl.org/constanttime.html>, 2018.

- 29 Thomas Pornin. Bearssl is an implementation of the ssl/tls protocol (rfc 5246) written in c. <https://bearssl.org>, 2022.
- 30 Fran ois Pottier and Vincent Simonet. Information flow inference for ml. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 319–330, 2002.
- 31 Jakob Rehof et al. Tractable constraints in finite semilattices. *Science of Computer Programming*, 35(2-3):191–221, 1999.
- 32 Bruno Rodrigues, Fernando Magno Quint o Pereira, and Diego F. Aranha. Sparse representation of implicit flows with applications to side-channel detection. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 110–120, New York, NY, USA, 2016. ACM. doi:10.1145/2892208.2892230.
- 33 Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.
- 34 Georgios Sakkas, Madeline Endres, Benjamin Cosman, Westley Weimer, and Ranjit Jhala. Type error feedback via analytic program repair. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 16–30, 2020.
- 35 Luigi Soares and Fernando Magno Quint o Pereira. Memory-safe elimination of side channels. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 200–210. IEEE, 2021.
- 36 Juraj Somorovsky. Curious padding oracle in openssl (cve-2016-2107), 2016. Last Retrieved: Jan 2024. URL: <https://web-in-security.blogspot.co.uk/2016/05/curious-padding-oracle-in-openssl-cve.html>.
- 37 Juraj Somorovsky. Systematic fuzzing and testing of tls libraries. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS ’16, pages 1492–1504, New York, NY, USA, 2016. ACM. doi:10.1145/2976749.2978411.
- 38 Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, 1996.
- 39 G Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. *ACM Sigplan Notices*, 39(11):85–96, 2004.
- 40 Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266. ACM, 2016.
- 41 TrustedFirmware. Mbed tls. <https://www.trustedfirmware.org/projects/mbed-tls>, 2022.
- 42 Dennis Volpano and Geoffrey Smith. A type-based approach to program security. In *TAP-SOFT’97: Theory and Practice of Software Development: 7th International Joint Conference CAAP/FASE Lille, France, April 14–18, 1997 Proceedings* 22, pages 607–621. Springer, 1997.
- 43 Guanhua Wang, Sudipta Chatopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead defense against spectre attacks via program analysis. *IEEE Transactions on Software Engineering*, 47(11):2504–2519, 2019.
- 44 Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. Identifying {Cache-Based} side channels through {Secret-Augmented} abstract interpretation. In *28th USENIX security symposium (USENIX security 19)*, pages 657–674, 2019.
- 45 Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. Cached: Identifying cache-based timing channels in production software. In *Proceedings of the 26th USENIX Security Symposium*, pages 235–252, 2017.
- 46 Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. Eliminating timing side-channel leaks using program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 15–26, 2018.
- 47 Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: a timing attack on openssl constant-time rsa. *Journal of Cryptographic Engineering*, 7:99–112, 2017.
- 48 Anna Zaks and Amir Pnueli. Covac: Compiler validation by program analysis of the cross-product. In *International Symposium on Formal Methods*, pages 35–51. Springer, 2008.

- 49 Danfeng Zhang, Aslan Askarov, and Andrew C Myers. Language-based control and mitigation of timing channels. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 99–110, 2012.
- 50 Danfeng Zhang and Andrew C Myers. Toward general diagnosis of static errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 569–581, 2014.
- 51 Danfeng Zhang, Yao Wang, G Edward Suh, and Andrew C Myers. A hardware design language for timing-sensitive information-flow security. *ACM SIGPLAN Notices*, 50(4):503–516, 2015.

Defining Name Accessibility Using Scope Graphs

Aron Zwaan 

Delft University of Technology, The Netherlands

Casper Bach Poulsen 

Delft University of Technology, The Netherlands

Abstract

Many programming languages allow programmers to regulate *accessibility*; i.e., annotating a declaration with keywords such as `export` and `private` to indicate where it can be accessed. Despite the importance of name accessibility for, e.g., compilers, editor auto-completion and tooling, and automated refactorings, few existing type systems provide a formal account of name accessibility.

We present a declarative, executable, and language-parametric model for name accessibility, which provides a formal specification of name accessibility in Java, C#, C++, Rust, and Eiffel. We achieve this by defining name accessibility as a predicate on *resolution paths* through *scope graphs*. Since scope graphs are a language-independent model of name resolution, our model provides a uniform approach to defining different accessibility policies for different languages.

Our model is implemented in Statix, a logic language for executable type system specification using scope graphs. We evaluate its correctness on a test suite that compares it with the C#, Java, and Rust compilers, and show we can synthesize access modifiers in programs with holes accurately.

2012 ACM Subject Classification Software and its engineering → Compilers; Software and its engineering → Language features; Theory of computation → Program constructs

Keywords and phrases access modifier, visibility, scope graph, name resolution

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.47

Related Version *Extended Version:* <https://doi.org/10.48550/arXiv.2407.09320> [36]

Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact):* <https://doi.org/10.4230/DARTS.10.2.27>

Acknowledgements We thank Friedrich Steimann for challenging us to specify and formalize access modifiers using scope graphs, and the anonymous reviewers for their helpful comments.

1 Introduction

Many programming languages, especially object-oriented ones, support *information hiding*, i.e., regulating from which positions in a program a declaration can be accessed. Information hiding is used to enforce invariants of particular code units, implement design patterns (e.g. the singleton pattern), improve modularization, limit public APIs to offer guidance to library users and guarantee forward compatibility. Support for information hiding is usually provided using *access modifier keywords*¹ (*access modifiers* for short), such as `public`, `protected`, `internal` and `private`. Each of these corresponds with a particular accessibility policy that is validated by the type checker.

Although recent research has not paid much attention to access modifiers, there are still good reasons to study their semantics. First, understanding access modifiers is required to implement (alternative) compilers and editor services correctly. In particular, disregarding accessibility may result in incorrect name binding, and hence incorrect program behavior. Second, formalizing access modifiers enables reasoning about the meaning of programs.

¹ Other common names include “access specifier” or “visibility modifier”.

47.2 Defining Name Accessibility Using Scope Graphs

```

package p1;
class A {
    int x;
}

package p2;
class B extends p1.A { }

package p1;
class C extends p2.B {
    int y = x;
}

```

(a) Inheritance through Packages.

```

package p;
class A {
    protected int x;
}

class B extends A {
    private int x;
}

class C extends B {
    int y = x;
}

```

(b) Inaccessible or Shadowed?

```

package p;
class A {
    private int x = 0;
    protected int y = 1
}

class B {
    int x = 3;
    int y = 4;
    class C extends A {
        int z = x + y
    }
}

```

(c) Accessibility and Shadowing.

 **Figure 1** Examples of intricate Access Modifier semantics. Classes are assumed to be public.

Finally, program transformation tools, such as automated refactorings, must handle the semantics of accessibility correctly. This is especially relevant for research on large-scale automated transformations, aimed at dealing with large (legacy) codebases. It is often infeasible to check transformations performed with such tools manually. Thus, the correctness of these transformations must be guaranteed through other means.

The meaning of access modifiers can be intricate in corner cases. We illustrate that using the examples in Figure 1. In Figure 1a, there is an inheritance chain, where class C extends class B, which itself extends A. Classes A and C reside in package p1, while B is in p2. Class A defines a package-accessible field x, which is accessed in C. The question here is whether that access is actually allowed. One could reason that it is correct, as the access occurs in the same package as the declaration, so a package-level declaration should be visible. On the other hand, one could consider x not inherited by B [11, §8.2], and thus not inherited by C either. In fact, the Java language designers chose the second option, rejecting this program [23, §4.2]. Using ((A) `this`).x is accepted however.

Something similar happens in Figure 1b. Here, one can consider the reference x in class C to be invalid, as the field in class B is inaccessible. Alternatively, under the assumption that B.x is *out of scope*, the reference can be valid, pointing to A.x. In this case, Java checks accessibility *after shadowing*, so this program is again rejected. However, in Figure 1c, accessibility does influence the binding. The reference x binds to the field of the *enclosing* class B, as the field inherited from class A is inaccessible. However, reference y binds to the field inherited from A. Thus, in this case, the *accessibility* of the inherited fields determines the resolution of x and y; i.e., accessibility is checked *before shadowing*. This shows that specifying accessibility is essential to defining the name binding of a language correctly.

Unintuitive semantics of accessibility occurs in non-object-oriented languages as well. For example, the accessibility scheme of Agda seems simple: definitions are either public or module-private, and imported definitions can be re-exported. However, issue #5461² reports that re-exports in a private block are still exposed to the outside world. While this intuitively seems wrong to most commenters, an argument is made that this is actually the intended behavior. The discussion stalls shortly after a remark that talking about intended behavior is “meaningless without a specification”.

² <https://github.com/agda/agda/issues/5461>

These examples show that the meaning of access modifiers is not always obvious. Hence, language designers should define their semantics unambiguously. Ideally, that is done through *specifications* containing *inference rules*. Inference rules allow unambiguous interpretation of the meaning of programming language constructs, including name binding. However, perhaps surprisingly, a general model for defining access modifiers has never been proposed.

Perhaps closest is the work of Steimann and Thies [25] (later incorporated in the JRRT refactoring tool [23]). They propose a constraint-based approach to automating refactorings in Java, by collecting and solving *accessibility constraints*. These constraints are generated using *constraint generation rules*, which cover the access rules the Java compiler enforces. By solving these constraints, changes in accessibility implied by the refactoring can be inferred, yielding type- and behavior-preserving refactorings.

Steimann and Thies' work solves the problem of making refactorings in Java sound regarding accessibility. However, it does not yet give a high-level explanation of the meaning of access modifiers. This is partly because the constraint generation rules need several low-level details to catch some intricate corner cases, but also because the function that computes the minimal required accessibility level is not given, as it was “unpleasant to specify” and “of no theoretical interest” [25, §5.2]. Therefore, their work cannot easily be adapted to a different language or a different application (e.g., a type checker).

To advance the state of the art, we pursue the following goals:

- Explain the meaning of access modifiers.
- Explain the (subtle) differences between access modifiers in different languages.
- Provide a framework for experimenting with feature combinations that do not (yet) exist in other languages.

To this end, we do not fully formalize one particular language, but rather define a toy language that incorporates and combines a large number of accessibility features. To abstract over low-level name resolution details, we use *scope graphs* [18, 27, 22, 38]. In this paper, we demonstrate this is a natural fit, because accessibility can be expressed as a predicate over paths in a scope graph. The specification is written in the logic language Statix [27, 22], which has a well-defined declarative semantics and also supports generating executable type-checkers automatically.

We compare these executable type checkers with reference compilers of Java, C#, and Rust, showing that we accurately captured the semantics of access modifiers in some real-world languages. Moreover, using Statix/scope graphs as a basis for (*language-parametric*) *refactorings* is an active topic of research [16, 29, 15, 3]. We envision that this will provide accessibility-aware refactorings similar to Steimann et al., without requiring significant additional effort. This is substantiated by the fact that Statix-based code completion [19] proposes an access modifier if and only if it would not cause accessibility errors elsewhere in the program.

In summary, the contributions of this paper are as follows:

- We provide a systematic classification of accessibility features (Section 2);
- we apply our taxonomy to Java, C++, C#, Rust, and Eiffel (Section 2);
- we present a specification of (various versions of) accessibility on modules (Section 5), subclasses (Section 6), and their conjunctive and disjunctive combination (Section 7);
- we extend our specification with accessibility-restricting inheritance (Section 8);
- we prove some theorems about our model, showing it is well-behaved (Section 9); and
- we implement our specification in Statix, and compare it with the standard compilers of Java, C#, and Rust. Moreover, we show access modifiers can be synthesized accurately using Statix-based Code Completion [19] (Section 10).

This paper comes with an artifact that allows reproducing the evaluation [35], and appendices containing a full specification of the access modifiers and proofs of the stated theorems [36].

2 Access Modifiers in Real-World Languages

In this section, we explore the design space of access modifiers as they occur in real-world languages. We first motivate why languages have access modifiers (Section 2.1). After that, we discuss common accessibility features (Section 2.2), summarizing them in a feature model (Section 2.3).

2.1 Why Accessibility?

Most programming languages allow programmers to define entities (variables, functions, types, etc.), and assign a name to them. That name can then be used to refer to the introduced entity from other positions in the program. However, as there is typically a large number of entities within a software project, most languages offer a notion of modularization to group related definitions. Equally named definitions in different modules can be distinguished by qualifying them with the name of the module in which they reside. Unqualified (or partially qualified) names by default resolve within their enclosing module, or imported modules. Details of this scheme differ from language to language, but generally aim to make definitions easy to refer to (e.g., by minimizing the number of required qualifiers), while trying to be unambiguous to the compiler and the programmer.

However, these rules may often be too lenient with respect to the intention of the programmer. A definition may be accessible from scopes where it is not intended to be used. This can have detrimental effects on the quality of a software artifact. For example, exposing all internal definitions of a library makes it (1) less intuitive to its users, (2) prone to forward compatibility issues and technical debt (e.g. strong coupling).

For these reasons, many programming languages provide constructs that give *the programmer* control over the regions of code where a definition can be accessed. For example, in many object-oriented languages, a class can access fields from its ancestor classes by default (language-controlled). However, if the programmer does not want a field to be accessible from subclasses, they can add a `private` access modifier. This modifier *prevents* access from all other classes (programmer-controlled). Although many constructs that provide access control to the programmer can be envisioned, most languages settle on a limited set of keywords that can be attached to a definition. In practice, this relatively simple scheme has proven powerful enough to cover most use cases.

2.2 Accessibility in Practice

Next, we explore how languages typically provide modularization and accessibility features.

Modules. A common feature that provides modularization is *modules* (also called “package” or “namespace”). A module is a syntactic construct that introduces a named collection of definitions. Members of modules can be accessed using the name of the module, for example in a preceding import statement, or as a qualifier to the name of the member that is accessed.

Hiding a definition from other modules is the simplest accessibility restriction that can be applied with respect to modules. For example, Java declarations without an access modifier can only be accessed within the same package. Rust items without a modifier behave similarly, except that declarations can still be accessed from submodules.

Some languages have multiple notions of modularization. For example, C# has assemblies, namespaces, and files, where a namespace can comprise multiple files, and/or a file can contain multiple namespaces. The `internal` keyword in C# restricts accessibility to the *assembly*, and the `file` keyword (introduced in C# 11 [32]) to the current file. Similarly, Java 9 introduces *modules* [21], with features to restrict access from external modules.

```

mod outer {
    mod inner {
        pub x = 42;
    }
    pub use inner::x;
}

fn main() {
    // ERROR: inner is inaccessible:
    // let x = outer::inner::x;
    let x = outer::x;
    println!("{}")
}

```

Figure 2 Re-exports can change Accessibility.

Some languages give some more control over *which* modules a declaration can be accessed from. For example, Rust has the `pub(in path)` access modifier, where *path* refers to some enclosing module. This enables programmers to expose items to an arbitrary ancestor.

Imports usually do not affect the visibility of a declaration. A notable exception to this rule is *re-exporting* (e.g., as implemented in Rust), which can actually *change* the visibility of a declaration, as shown in Figure 2. In this program, the module `inner` is accessible in `outer`, but not in its parent (the root scope). Therefore, the function `main` cannot access its field `x`. However, `outer` re-exports `inner::x`, which gives rise to a new definition `outer::x`. As `outer` is accessible in the root scope, so is this definition. Hence, via the re-export, `main` can access `x`, although the original declaration was hidden.

From an accessibility point of view, re-exporting can typically be considered as a combination of an import and a declaration, where the declaration always points to the imported member. The re-exported item (`inner::x` in the example) should be accessible from the location of the *re-export*. References to the re-export should have access to the location of the re-export, but not necessarily to the location of the original declaration. In fact, for any access path, it does not matter whether the declaration is a re-export or not.

Classes. A special modularization concept is the notion of *classes*, which represent composite data types with associated operations (methods). Where simple modules only have a static interpretation, an arbitrary number of class instances can exist at runtime.³ While modules can implicitly be related to each other by their relative position, such a relation does not exist for classes. However, classes can extend other classes, ensuring the subclass inherits the fields of its parent class. This creates an inheritance hierarchy orthogonal to the module hierarchy.

Object-oriented languages usually provide modifiers to control accessibility over the inheritance chain. For example, Java and C# have a `private` keyword, which prevents access outside the defining class. Additionally, the `protected` keyword allows access from subclasses, but prevents access from any other location.

In Java and C#, the accessibility level is inherited with the field. That means, if a field in the superclass is `protected`, it will be protected in the subclass as well. However, C++ allows restricting the accessibility of members of the parent class. A `private` modifier on extends-clauses will make all inherited public/protected members private on instances of the subclass. Similarly, a `protected` modifier will make all inherited public members protected.

Finally, some languages allow specifying “friend” classes, which grant the friend access to its members. This enables fine-grained access control, independent from module and class hierarchies. While discouraged in C++, Eiffel provides only this access control mechanism.

³ At this point, we slightly over-simplify the reality. For example, neither parameterized modules (ML) nor `objects` (e.g. Scala/Kotlin) fit in this scheme. We made this choice deliberately, to cover the most prevalent cases. We conjecture that the techniques we develop for classes can be applied to parameterized modules (and vice versa for modules and objects) but leave explicating that to future work.

Interaction. Accessibility restrictions on modules and classes can be combined. This is very explicit in C#, which has `protected internal` and `private protected` as additional modifiers. The former permits access from within the assembly (similar to `internal`) *and* to subclasses (similar to `protected`), even if they live outside the assembly. Analogously, `private protected` grants access to subclasses in the same assembly only, which is equivalent to the conjunction of `internal` and `protected`.

2.3 Classification

These concepts are organized and related in the feature model in Figure 3. Following the previous discussion, the main features are modules and classes. We have only a single feature for modules, because the different variants are (apart from C#'s files and namespaces) typically not mutually nested. The `internal` keyword can either relate to the containing module (Direct) or an arbitrary parent module (Ancestor). We explore this further in Section 5.

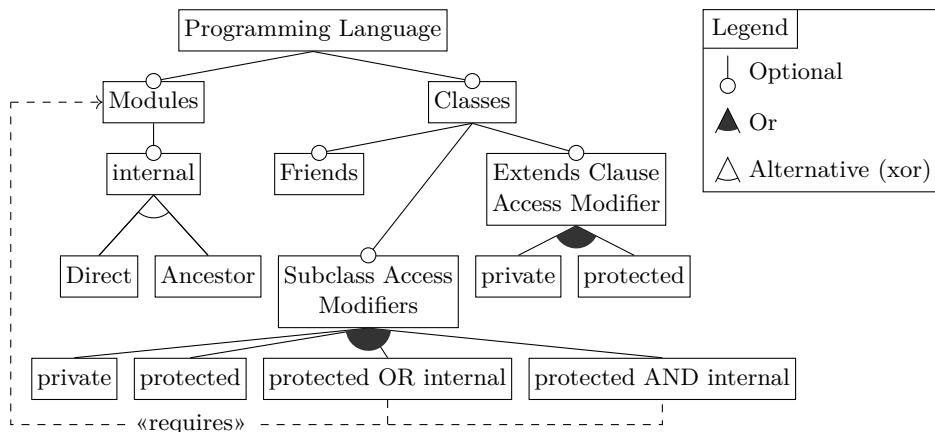


Figure 3 Feature Model for Access Control.

Table 1 Languages classified according to the feature model in Figure 3.

	Java	C#	C++	Eiffel	Rust
Modules	✓	✓	✓		✓
Internal	Direct	Direct ⁴	Direct		Ancestor
Classes	✓	✓	✓	✓	
Friends			✓	✓	
Subclass Acc. Mod.	✓	✓	✓	✓	
<code>private</code>	✓	✓	✓	✓	
<code>protected</code>		✓	✓		
<code>protected internal</code>	✓	✓			
<code>protected & internal</code>		✓	✓		
Extends Clause Acc. Mod.			✓		
<code>private</code>			✓		
<code>protected</code>			✓		

In the Classes category, the three subfeatures denote the three mechanisms for access control: Friends allow access to other classes by name, Subclass Access Modifiers are access modifiers on definitions that determine how it is accessible within the class hierarchy (Sections 6 and 7), and Extends Clause Access Modifiers (Section 8) are access modifiers on extends clauses, as seen in C⁺⁺. The latter two have subfeatures for each concrete keyword associated with the access control mechanism. For that reason, `private` and `protected` occur twice: once on definitions and once on extends clauses. Table 1 classifies several languages according to this scheme. In the remainder of this paper, we develop AML (Access Modifier Language), a language that covers all features. To this end, we first introduce scope graphs (Section 3), and a base language for AML (Section 4).

3 Using Scope Graphs to Model Name Binding in Programs

In the previous section, we sketched the landscape of access modifiers. This discussion was based largely on prose specifications as well as experiments with compiler implementations. No language specification we are aware of provides a more rigorous model of accessibility (or even non-lexical name binding). In this section, we introduce *scope graphs* [18, 27, 22, 38], and argue that they provide a suitable framework for such a model. Section 4 introduces AML (Access Modifier Language), a toy language with a type system defined using scope graphs. Sections 5–8 will extend this language with all accessibility features from Figure 3.

3.1 Scope Graphs as A Model for Name Binding

From a name binding perspective, classes and modules have some similarities. Each of these constructs can be thought of as introducing a “scope” (region of code), in which declarations live, and in which names can be resolved. Scopes are related to each other in various ways. First, modules are related according to their relative position in the abstract syntax tree. In addition, imports and extends clauses relate arbitrary modules and classes, respectively. Resolving a reference corresponds to finding a matching declaration in a scope that is reachable from the scope of the reference. For example, a reference may resolve to a declaration if it lives in a lexically enclosing scope, or in a module that is imported in an enclosing scope.

Scope graphs [18, 27, 22, 38] make this more precise. In this model, the name binding structure of a program is represented by a graph. Figure 4 (adapted from Poulsen et al. [20, Fig. 1]) gives an example program and its corresponding scope graph. A scope is represented by a circular node in the graph. For example, s_0 represents the global scope, and s_A , s_B and s_C represent the bodies of modules A, B, and C, respectively. Scopes are related using labeled, directed edges. For example, s_A is lexically enclosed by s_0 , and thus the graph contains an edge from s_A to s_0 with label `LEX`. Similarly, s_B imports s_A , and thus the graph contains an edge $s_B \xrightarrow{\text{IMP}} s_A$. Finally, scope graphs contain declarations. For example, a declaration of `i` in scope s_C is represented by the $s_C \xrightarrow{\text{VAR}} i : \text{int}$ edge/node pair. Similarly, the modules are declared in the root scope (e.g., $s_0 \xrightarrow{\text{MOD}} A \sim s_A$). The language specification determines which data is included in the declaration. Similarly, the labels for edges and declarations can be chosen to match the (binding) constructs of the language.

⁴ Either the most direct enclosing *file* (`file`), or most directly enclosing *assembly* (`internal`), possibly bypassing some namespaces.

47:8 Defining Name Accessibility Using Scope Graphs

```
module A {
    var i = 5
}
module B {
    import A
}
module C {
    import B
    var j = i
}
```

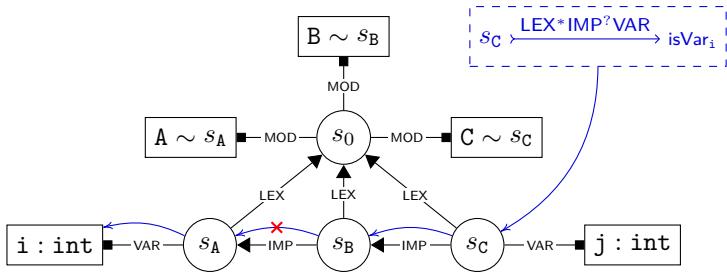


Figure 4 Reachability example. The $\text{IMP}^?$ part in the regular expression prevents traversal over the second IMP edge.

```
module D {
    var x = 3
    module E {
        import F
        var y = x
    }
}
module F {
    var x = 4
}
```

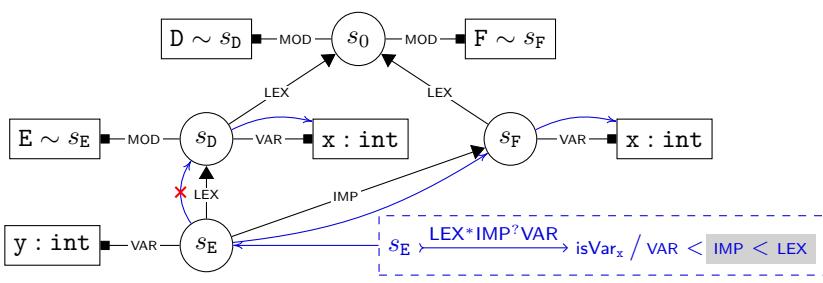


Figure 5 Shadowing example. The highlighted label order causes the edge to s_F to have priority.

Reachability. To resolve a reference, a *query* is executed to find a valid path in the scope graph from the scope of the reference to a matching declaration. Queries give specification writers several options to filter paths, to retain only valid paths. First, a unary predicate selects valid declarations. Usually, this predicate matches declarations with the name of the reference. Second, a *regular expression* over labels is used to select valid paths. This regular expression can, for example, be used to prevent transitive imports, or accessing members in a lexical parent of an imported module.

Figure 4 illustrates this with the query for i in module C (dashed blue box). The parameter on the arrow ($\text{LEX}^*\text{IMP}^?\text{VAR}$), is a regular expression that defines which paths to declarations are valid. The LEX^* indicates that a path may traverse an arbitrary number of LEX -edges. This corresponds to looking for variables in enclosing scopes. Next, the $\text{IMP}^?$ part indicates that zero or one IMP -edges can be traversed. Finally, the regular expression ends with VAR to ensure all paths resolve in variable declarations only, excluding e.g. modules. The isVar_i parameter matches all variable definitions with name i (isVar is defined in the next section). The candidate path (shown as blue edges) does not match this regular expression. Because IMP -labeled edges may only be traversed one time, the step to s_A cannot be made. In other words: the declaration of i in A is not *reachable* from C .

Visibility. Not every declaration that is reachable (i.e., for which a valid access path exists) can actually be referenced, due to *shadowing*. For example, in most languages, local definitions have higher priority than imported ones. We call reachable declarations that are not shadowed by any other declaration *visible*.

In scope graphs, visibility can be encoded using a partial order on labels. For example, an order $\text{VAR} < \text{IMP}$ encodes that (local) variable declarations shadow imported declarations. This is illustrated in Figure 5. The reference x in module F can refer to the declaration in

module D as well as the one in module E. Because the label order (third argument) indicates that imports shadow lexically enclosing scopes ($\text{IMP} < \text{LEX}$). Thus, the variable resolves to the declaration in s_F . Alternatively, if $\text{LEX} < \text{IMP}$, it would resolve to x in s_D . Finally, if neither $\text{LEX} \not\prec \text{IMP}$ nor $\text{IMP} \not\prec \text{LEX}$, both paths would be included in the query result.

In summary, scope graphs model the name binding structure of a program using nodes for scopes and declarations, and edges for relations between those. Queries can be used to model reference resolution. A query selects a declaration when (1) it matches some predicate, and (2) there exists a path to it of which the labels match a regular expression, and (3) no other paths that traverse labels with higher priority exist. The result of a query is a set of paths that lead to these matching declarations.

Accessibility. We can model extensibility using plain scope graphs by including accessibility information in the *declaration*. In other words, a declaration of a variable in a scope graph contains not only a name and a type, but also its accessibility level. *After resolution*, we check if the path that the query returns is actually valid according to the accessibility level of the declaration. For example, if a variable is private, but an `EXT`-edge (for class *extension*) is traversed, an error is emitted. With this pattern, we can model all accessibility features.

Notation. Figures 4 and 5 introduce the graphical notation of scope graphs. In text, variable s ranges over scopes, and S over sets of scopes. Moreover, we use the following notation for assertions on scope graphs: $s_1 \xrightarrow{L} s_2 \in \mathcal{G}$ means “scope graph \mathcal{G} has an L-labeled edge from s_1 to s_2 ”, and $s \xrightarrow{D} d \in \mathcal{G}$ means that \mathcal{G} has a declaration with data d under label D in scope s . Moreover, we write queries in the following way:

$$\text{query}_{\mathcal{G}} s \xrightarrow{R} \mathcal{P} / \mathcal{O} \mapsto R$$

where \mathcal{G} is the scope graph in which the query is resolved, s is the scope in which the resolution starts, R is the regular expression that paths must adhere to, and \mathcal{P} is the predicate that declarations must match. \mathcal{O} is the strict partial order on labels used for shadowing. It is usually written as $L_1 < L_2 < \dots < L_n$. We omit the label order when there is no shadowing. R is the result set containing tuples of paths and declarations. When we expect a single result, we use $\{(p, d)\}$ to match on the value in the set. Paths are alternating sequences of scopes and labels, written as $s_1 \xrightarrow{L_1} s_2 \dots s_m$. Paths do not include the declaration it resolved to, but stop at the scope in which the declaration occurs. The functions `src(p)`, `tgt(p)` refer to the source and target scope of a path, respectively. `scopes(p)` denotes all scopes in a path.

4 AML: The Base Language

In the next sections, we show how scope graphs support intuitive formalization of accessibility. We will do so by defining *AML* (Access Modifier Language). The base syntax (which will be extended later) is given in Figure 6. In AML, a program consists of a list of modules. Each module can define other modules, import other modules, and contain class definitions. A class can optionally extend another class, and contains a list of field declarations. Each field has an access modifier, and is initialized by some expression. Possible expressions include references, integer constants, class instance creation, field access, and binary operations.

At the right-hand side of Figure 6, the scope graph parameters are shown. There are three labels that connect scopes. `LEX` denotes lexical scoping, `IMP` denotes imports, and `EXT` class extension. The other three labels are used for declarations. `MOD` is used for module declarations, `CLS` for classes, and `VAR` for variables/fields. Next, we assume that each *module*

47:10 Defining Name Accessibility Using Scope Graphs

$\langle \text{prog} \rangle ::= \langle \text{mod} \rangle^*$	$\langle l \rangle ::= \text{LEX} \mid \text{IMP} \mid \text{EXT}$
$\langle \text{mod} \rangle ::= \text{module } \langle x \rangle \{ \langle \text{md} \rangle^* \}$	$ \text{MOD} \mid \text{CLS} \mid \text{VAR}$
$\langle \text{md} \rangle ::= \langle \text{mod} \rangle \mid \text{import } \langle x \rangle \mid \langle \text{cls} \rangle$	$ \text{THIS}_M \mid \text{THIS}_C$
$\langle \text{cls} \rangle ::= \text{class } \langle x \rangle (:\langle \text{acc} \rangle \langle x \rangle)^? \{ \langle \text{cd} \rangle^* \}$	$\langle d \rangle ::= \text{mod } \langle x \rangle : \langle s \rangle$
$\langle \text{cd} \rangle ::= \langle \text{acc} \rangle \text{ var } \langle x \rangle = \langle e \rangle \mid \langle \text{cls} \rangle$	$ \text{cls } \langle x \rangle : \langle s \rangle$
$\langle \text{acc} \rangle ::= \text{public} \mid \dots$	$ \text{var } \langle x \rangle : \langle T \rangle @ \langle A \rangle$
$\langle e \rangle ::= \langle n \rangle \mid \langle x \rangle \mid \text{new } \langle x \rangle () \mid \langle e \rangle . \langle x \rangle \mid \dots$	$ \langle s \rangle$
	$\langle T \rangle ::= \text{int} \mid \text{inst } \langle s \rangle$
	$\langle A \rangle ::= \text{PUB} \mid \dots$

Figure 6 Syntax of AML. The highlighted positions indicate extensions in later sections. The syntax of the complete language can be found in Appendix A [35].

Data Matching Predicates

$$\begin{array}{ll} \text{isMod}_x(\text{mod } x' : s) \Leftarrow x = x' & \text{isCls}_x(\text{cls } x' : s) \Leftarrow x = x' \\ \text{isVar}_x(\text{var } x' : T @ A) \Leftarrow x = x' & \text{isScope}_s(s') \Leftarrow s = s' \end{array}$$

Class Members

$$\boxed{\mathcal{P}(d)}$$

$$\frac{\text{D-DEF} \quad s \vdash_{\mathcal{G}} e : T \quad s \vdash_{\mathcal{G}} \text{acc} \Rightarrow A \quad s \xrightarrow{\text{VAR}} (\text{var } x : T @ A) \in \mathcal{G}}{s \vdash_{\mathcal{G}} \text{acc var } x = e \text{ OK}}$$

Type of Expression

$$\boxed{s \vdash_{\mathcal{G}} e : T}$$

$$\frac{\text{T-VAR} \quad \text{query}_{\mathcal{G}} s \xrightarrow{\text{LEX}^* \text{EXT}^* \text{VAR}} \text{isVar}_x / \text{VAR} < \text{EXT} < \text{LEX} \mapsto \{ \langle p, \text{var } x : T @ A \rangle \} \quad s \vdash_{\mathcal{G}} p ! A}{s \vdash_{\mathcal{G}} x : T}$$

$$\frac{\text{T-FLD} \quad \text{query}_{\mathcal{G}} s_c \xrightarrow{\text{EXT}^* \text{VAR}} \text{isVar}_x / \text{VAR} < \text{EXT} \mapsto \{ \langle p, \text{var } x : T @ A \rangle \} \quad s \vdash_{\mathcal{G}} p ! A}{s \vdash_{\mathcal{G}} e.x : T}$$

Access Modifier

$$\boxed{s \vdash_{\mathcal{G}} \text{acc} \Rightarrow A}$$

Access Policy

$$\boxed{s \vdash_{\mathcal{G}} p ! A}$$

$$\text{A-PUB} \frac{}{s \vdash_{\mathcal{G}} \text{public} \Rightarrow \text{PUB}}$$

$$\text{AP-PUB} \frac{}{s \vdash_{\mathcal{G}} p ! \text{PUB}}$$

Module and Class References

$$\boxed{s \vdash_{\mathcal{G}} x \xrightarrow{M} s_m \quad s \vdash_{\mathcal{G}} x \xrightarrow{C} s_c}$$

$$\frac{\text{Q-MOD} \quad \text{query}_{\mathcal{G}} s \xrightarrow{\text{LEX}^* \text{MOD}} \text{isMod}_x / \text{MOD} < \text{LEX} \mapsto \{ \langle p, \text{mod } x : s_m \rangle \}}{s \vdash_{\mathcal{G}} x \xrightarrow{M} s_m}$$

$$\frac{\text{Q-CLS} \quad \text{query}_{\mathcal{G}} s \xrightarrow{\text{LEX}^* \text{IMP}^* \text{CLS}} \text{isCls}_x / \text{CLS} < \text{IMP} < \text{LEX} \mapsto \{ \langle p, \text{cls } x : s_c \rangle \}}{s \vdash_{\mathcal{G}} x \xrightarrow{C} s_c}$$

Figure 7 Typing Rules of AML. Accessibility is integrated at the highlighted positions. The full type system specification can be found in Appendix A [35].

scope has a THIS_M edge pointing to itself, and similarly, each class has a THIS_C scope pointing to itself. This will be used to resolve enclosing classes or modules. The sort $\langle d \rangle$ denotes the data that can be associated with scopes. Modules and classes are characterized by their name and the scope of their body. A field has a name, a type $\langle T \rangle$, and an accessibility level $\langle A \rangle$. Scopes that are not declarations implicitly map to themselves. To query declarations, we use the four predicates shown at the top of Figure 7, which each match a single kind of declaration. Depending on the type of access control we formalize, different access modifiers will be used. Therefore, we have left the $\langle acc \rangle$ and $\langle A \rangle$ productions partially unspecified. Each section will instantiate those appropriately.

Typing Rules. Figure 7 presents some typing rules of AML. The rules are written in a declarative style, where a scope graph \mathcal{G} that models the program is assumed. Constraints over the scope graph are used as premises. The highlighted premises show where accessibility is integrated into the type system. We now discuss each of the presented rules.

The **D-DEF** rule asserts a declaration is well-typed if the initialization expression e has some type T (first premise), the access modifier acc corresponds to some accessibility policy A (second premise), and an appropriate declaration exists in the scope graph (third premise). The accessibility policy is included in the declaration, which enables us to validate accessibility when type checking references.

Next, rule **T-VAR** defines how references are type checked in a current scope s . First, it performs a query that looks into the lexical context (LEX^*), parent classes (EXT^*), and eventually resolves to a variable declaration (VAR). It matches only variables with the same name as the reference (isVar_x). Regarding shadowing, it prefers local variables over variables from a parent class ($\text{VAR} < \text{EXT}$), and variables from parent classes over variables from enclosing classes ($\text{EXT} < \text{LEX}$). The query should return a single result, as the name would otherwise be ambiguous. From this result, the access path p , type T , and accessibility policy A are extracted. The path and the accessibility policy are used in the second (highlighted) premise ($(s \vdash_{\mathcal{G}} p ! A)$), which asserts that “accessibility policy A grants access via path p in scope s ”. In future sections, we will define new accessibility policy rules, that may prohibit access of a variable, even if the query premise resolved properly.

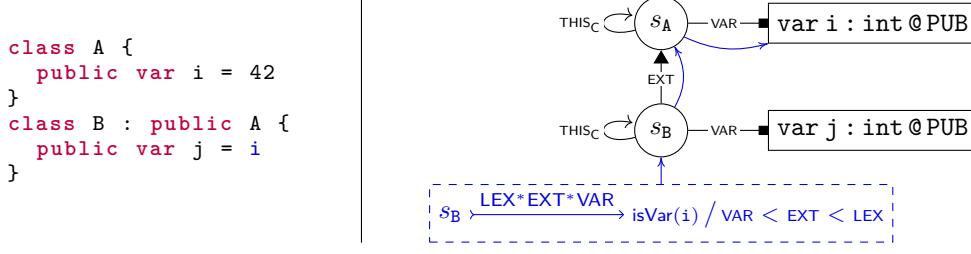
Note that, by having accessibility separated from the resolution, we do not capture the interaction between accessibility as shown in Figure 1c. We made this choice because the place where accessibility is integrated does not influence the access rules themselves, and this presentation allows more concise derivations, which makes the explanations more accessible. Appendix A.1 [35] shows how to integrate accessibility in the shadowing policy of a query, and is incorporated in the evaluation (Section 10).

For this base language, we only have the `public` access modifier. The **A-PUB** rule shows that this keyword corresponds to the PUB policy. The meaning of this policy is that access is allowed from any location, with any access path. This is encoded in the **AP-PUB** rule, which has no premises.

Finally, the last two rules define how references to classes and modules are resolved. Rule **Q-MOD** indicates that module reference x resolves to scope s_m if that scope is included in the closest module declaration with name x in the lexical context. Similarly, a class reference resolves to the scope of the closest class declaration s_c , preferring (non-transitively) imported classes over classes in the lexical context (**Q-CLS**).

Example. The example in Figure 8 shows two classes **A** and **B**. Both classes have a THIS_C -edge pointing to itself. Class **B** extends class **A**, which is represented by the $s_B \xrightarrow{\text{EXT}} s_A$ edge in the scope graph. Class **A** has a public field **i** with type `int`. The type as well as the corresponding

47:12 Defining Name Accessibility Using Scope Graphs



(a) Example program and (partial) scope graph.

$$\frac{\text{query}_G s_B \rightsquigarrow \text{isVar}_i / \dots \mapsto \{(s_B \xrightarrow{\text{EXT}} s_A, \text{var } i : \text{int} @ \text{PUB})\}}{s_B \vdash_G i : \text{int}}$$

(b) Part of typing derivation that shows how access is granted by the PUB accessibility policy.

Figure 8 Example AML program demonstrating the scope graph structure and name resolution with accessibility checking.

Enclosing Modules

$$\vdash_G s \uparrow_M S \quad \vdash_G s \uparrow_M s$$

$$\text{ENC-M} \frac{\text{query}_G s \xrightarrow{\text{LEX*THIS}_M} \top \mapsto R \quad S_M = \{s_m \mid \langle p_m, s_m \rangle \in R\}}{\vdash_G s \uparrow_M S_M}$$

$$\text{ENC-MI} \frac{\text{query}_G s \xrightarrow{\text{LEX*THIS}_M} \top / \text{THIS}_M < \text{LEX} \mapsto \{\langle p, s_m \rangle\}}{\vdash_G s \uparrow_M s_m}$$

Enclosing Classes

$$\vdash_G s \uparrow_C S \quad \vdash_G s \uparrow_C s$$

$$\text{ENC-C} \frac{\text{query}_G s \xrightarrow{\text{LEX*THIS}_C} \top \mapsto R \quad S_C = \{s_c \mid \langle p_c, s_c \rangle \in R\}}{\vdash_G s \uparrow_C S_C}$$

$$\text{ENC-Cl} \frac{\text{query}_G s \xrightarrow{\text{LEX*THIS}_C} \top / \text{THIS}_C < \text{LEX} \mapsto \{\langle p, s_c \rangle\}}{\vdash_G s \uparrow_C s_c}$$

Figure 9 Auxiliary relations for AML scope graphs.

PUB access policy are included in the scope graph declaration. Similarly, class B has a field j. The initialization expression of j references i, which is represented with the query shown in the dashed box.

Figure 8b shows the part of the typing derivation that checks the highlighted reference. Reference i is type checked in scope s_B, and has type int. The first premise repeats the query shown in the scope graph, with the parameters and result made explicit. In particular, the resolution path is $s_B \xrightarrow{\text{EXT}} s_A$. The validity of this path is checked by the second premise, which is satisfied by the AP-PUB rule.

Auxiliary Relations. Finally, Figure 9 presents some auxiliary relations that we will use later. First, the $\vdash_G s \uparrow_M S_M$ relation asserts that S_M is the set of scopes of the enclosing modules of s. It is defined as a query that looks for a THIS_M edge in the lexically enclosing

scopes. There is no shadowing, so R can contain multiple results in the case of multiple nested modules. The result R is translated to the set of module scopes by discarding the access paths.

This relation is inhabited for any enclosing module scope. The second relation $\vdash_{\mathcal{G}} s \uparrow_M s_m$ is only inhabited for the *innermost* enclosing module s_m . The query in its definition finds the closest THIS_M -edge, which is enforced by the shadowing policy $\text{THIS}_M < \text{LEX}$. Thus, the query returns only one result, from which the module scope s_m is extracted. Analogously, $\vdash_{\mathcal{G}} s \uparrow_C S_C$ relates s to all enclosing *class* scopes S_C , and $\vdash_{\mathcal{G}} s \uparrow_C s_c$ is satisfied if s_c is the *innermost* enclosing class of s .

5 Defining Module Visibility

Some languages have access modifiers that regulate the visibility of a declaration in other modules. For example, in Rust, it is possible to write `pub(in ...)` to indicate in which module a declaration is visible. Similarly, some languages support giving particular classes access to an item. It is the primary accessibility mechanism for Eiffel, and C++'s `friend` modifier enables this as well. Less flexible approaches, such as Java's package visibility and C#'s `internal` keyword can be seen as special instances of this mechanism.

To demonstrate how these access policies can be encoded using scope graphs, we extend our base language as follows. Figure 10a introduces an additional modifier keyword `internal`, which can contain references to modules. The declaration is visible in these modules only. The corresponding accessibility policy `MOD` has a set of scopes, each corresponding to a name given in the keyword argument.

Next, we explain how this keyword is interpreted. An `internal` declaration is accessible if the reference occurs in a module that the arguments to the `internal` modifier give access to. This is formalized in the rules given in Figure 10b. Rule A-INT translates an `internal` access modifier to the `MOD` policy. Each module name argument to the modifier (x_i) is resolved relative to the current scope s . This yields a collection of module scopes s_i , which

$$\langle acc \rangle ::= \dots | \text{internal} (\langle x \rangle^*) \quad \langle A \rangle ::= \text{MOD } S$$

(a) Syntax of `internal` keyword.

$$\text{A-INT} \frac{}{s \vdash_{\mathcal{G}} \text{internal} (\bar{x}_{0..n}) \Rightarrow \text{MOD } S} \quad \text{AP-INT} \frac{\vdash_{\mathcal{G}} s \uparrow_M S_M \quad s_m \in S_M \quad s_m \in S}{s \vdash_{\mathcal{G}} p ! \text{MOD } S}$$

(b) Semantics of `internal` keyword.

$$\text{A-INT}' \frac{\vdash_{\mathcal{G}} s \uparrow_M S_M}{s \vdash_{\mathcal{G}} \text{internal} (\bar{x}_{0..n}) \Rightarrow \text{MOD } S} \quad \text{AP-INT}' \frac{\vdash_{\mathcal{G}} s \uparrow_M s_m \quad s_m \in S}{s \vdash_{\mathcal{G}} p ! \text{MOD } S}$$

(c) Variant 1: Ancestor module only.

(d) Variant 2: Innermost module.

$$\text{AP-INT}'' \frac{\dots [\vdash_{\mathcal{G}} s \uparrow_M S_{M_i} \quad s'_m \in S_{M_i} \quad s'_m \in S]_{s' \in (\text{scopes}(p) \setminus \{ \text{tgt}(p) \})}}{s \vdash_{\mathcal{G}} p ! \text{MOD } S}$$

(e) Variant 3: Definition exposed to all classes in path.

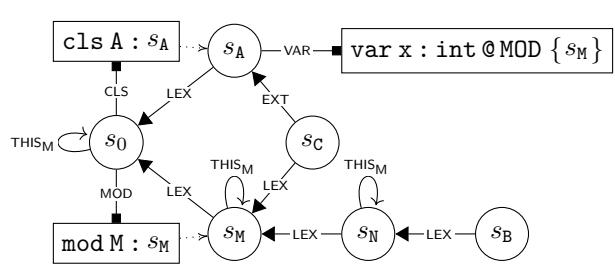
Figure 10 Extending AML (Figure 7) with module-level visibility.

47:14 Defining Name Accessibility Using Scope Graphs

```

class A {
    internal(M) var x = 42
}
module M {
    module N {
        class B {
            public var y =
                new C().x
        }
    }
    class C : public A { }
}

```



(a) Example program and partial scope graph demonstrating the `internal` access modifier.

$$\text{A-INT} \frac{s_A \vdash_G M \xrightarrow{M} s_M}{s_A \vdash_G \text{internal}(M) \Rightarrow \text{MOD}\{s_M\}}$$

(b) Part of typing derivation that shows how accessibility policy is derived.

$$\text{AP-INT} \frac{\dots \quad \vdash_G s_B \uparrow_M \{s_0, s_M, s_N\} \quad s_M \in \{s_0, s_M, s_N\} \quad s_M \in \{s_M\}}{s_B \vdash_G (s_C \xrightarrow{\text{EXT}} s_A) ! \text{MOD}\{s_M\}}$$

(c) Part of typing derivation that shows how access is granted by the `MOD` accessibility policy.

Figure 11 Example program demonstrating the meaning of the `internal` access modifier.

are included in the resulting policy. The `AP-INT` rule encodes that accessing an `internal` variable is valid if s_m , the scope of some enclosing module of s (the scope of the reference), is in the list of scopes to which access is granted.

Example. Figure 11 gives an example of an *internal* variable. Class A has a field x that can be accessed from module M. In the scope graph, this is indicated with the access policy `MOD {s_M}` on the corresponding declaration in s_A . The derivation of this policy is shown in Figure 11b. Module M contains a nested module N, which contains a class B. In class B, the field x is accessed on an instance of A. The (partial) typing derivation in Figure 11c shows this access is allowed by the `AP-INT` rule. The first premise asserts that s_0 , s_M and s_N are the enclosing modules of s_B . This can be seen in the scope graph, as those scopes are reachable via paths with regular expression `LEX*THIS_M` (Figure 9). As s_M occurs both in the enclosing modules and in the access policy, access is allowed.

Variant 1. Several variations on this scheme are conceivable. For example, languages can restrict the modules to which an `internal` modifier may expose a declaration. For example, Rust has the `pub(in <path>)` visibility modifier, similar to how we defined `internal`. However, at the $\langle path \rangle$ position, only “an ancestor module of the item whose visibility is being declared” is allowed [7, §12.6]. This is encoded in Figure 10c. Compared to `A-INT`, this rule adds premises (highlighted) that guarantee that the arguments of the `internal` modifier (x_i) resolve to an enclosing module ($s_i \in S_M$).

Note how these premises would make the example fail to type-check. Only s_0 is an enclosing module of s_A . In particular, the derivation in Figure 11b would have an additional premise $s_A \in \{s_0\}$, which is clearly unsatisfiable.

```

class A {
    private int x = 42;
    public int accessX(B b) {
        return b.x; // ERROR!
    }
}
class B extends A { }

class A {
    private int x = 42;
    public int AccessX(B b) {
        return b.x; // fine
    }
}
class B : A { }

```

Figure 12 Difference in `private` member access of subclass instances between Java and C#.

Variant 2. Next, consider the example in Figure 11a again. In the system above, `x` is accessible in `B`, because `x` is exposed to one of its enclosing modules (`M`). However, s_M is not its *innermost* enclosing module. Such a more lenient accessibility scheme might be desirable (e.g., Rust has this behavior), but languages such as Java do not allow this. To model these languages, we instead use the premise that asserts s_m is the *innermost* enclosing module scope. The rule for this variant is given in Figure 10d.

With this addition, the example would fail to type-check as well. The access validation (Figure 11c) would now have to satisfy $\vdash_G s_B \uparrow_M s_M$, which is impossible, as s_N is the innermost enclosing module.

Variant 3. Finally, consider example Figure 1a from the introduction again. In this example, the reference to `x` in class `C` was not valid, as `B` (by virtue of residing in a different package), did not inherit `x`. The (partial) rule in Figure 10e covers this case. For each scope in the path (apart from the target), it adds premises that assert that the definition is exposed to that scope (similar to s in Figure 10b).⁵ The target is excluded because it is not inheriting the accessed field, but rather defining it. (Recall that paths move from reference to declaration, so the target is the scope of the defining class.) For that reason, there is no need to assert it inherits the field.

When adding this rule fragment to the derivation in Figure 11c, there will be additional premises that validate that class `C` inherits `x`. This is the case, as `C` resides in module `M`.

6 Defining Subclass Visibility

Next, we consider how to define access modifiers that regulate access from other *classes*: the `private` modifier (Section 6.1), and the `protected` keyword (Section 6.2).

6.1 Private Access

The `private` access modifier is slightly challenging to define, as languages implement it differently. For example, C# allows accessing private variables on instances of *subclasses*, whereas Java does not. Consider the example programs in Figure 12. In the Java case, the access `b.x` is invalid, because it only allows access on instances of `A`.

On the other hand, Java exposes `private` members to the *outermost* enclosing class⁶, while C# only exposes members to the *defining* (i.e., innermost enclosing) class (including nested classes), as shown in Figure 13.

⁵ Alternatively, the premises of Figure 10d can be used when direct exposure is required.

⁶ “[When] the member or constructor is declared `private`, (...) access is permitted if and only if it occurs within the body of the top level class [sic!] that encloses the declaration of the member or constructor.” [11, §6.6.1]

47:16 Defining Name Accessibility Using Scope Graphs

```

class A {
    class B {
        private int x = 42;
    }
    int accessX(B b) {
        return b.x; // fine
    }
}

class A {
    class B {
        private int x = 42;
    }
    int AccessX(B b) {
        return b.x; // ERROR!
    }
}

```

Figure 13 Difference in `private` member access from enclosing class between Java and C#.

$$\langle acc \rangle ::= \dots \mid \text{private}\langle A \rangle ::= \dots \text{PRV} \quad \text{A-PRIV} \frac{}{s \vdash_{\mathcal{G}} \text{private} \Rightarrow \text{PRV}}$$

(a) Syntax of `private` keyword.

(b) `private` to PRV access policy.

$$\text{AP-PRIV} \frac{\vdash_{\mathcal{G}} s \uparrow_C S_C \quad \text{tgt}(p) \in S_C}{s \vdash_{\mathcal{G}} p ! A}$$

(c) Semantics of `private` keyword.

$$\text{AP-PRIV}, \frac{\dots \quad p \sim \text{LEX}^*}{s \vdash_{\mathcal{G}} p ! A}$$

(d) Prevent access on instances of subclasses.

$$\text{AP-PRIV''} \frac{\vdash_{\mathcal{G}} s \uparrow_C S_{C_{ref}} \quad \vdash_{\mathcal{G}} \text{tgt}(p) \uparrow_C S_{C_{decl}} \quad s_c \in S_{C_{ref}} \quad s_c \in S_{C_{decl}}}{s \vdash_{\mathcal{G}} p ! A}$$

(e) Allow access from enclosing classes.

Figure 14 Extending AML (Figure 7) with `private` visibility.

We start with modeling the C# semantics in Figures 14a–14c. Rule **AP-PRIV** states that the class in which the field is declared (which is the target of the path `tgt(p)`) should be an enclosing class of the scope in which the access occurs. This permits access from nested classes of `tgt(p)`, but does not expose it to enclosing classes. On the other hand, access on instances of subclasses is allowed, as there are no constraints on the structure of the path.

Note that we did not specify that this rule matches on the PRV policy specifically, but rather applies to *any* access policy A . This is a deliberate choice; it adds the possibility of using this rule as a fallback in case no other rule works. This ensures other accessibility policies will never be more strict than PRV, which corresponds to general intuition. By matching on an arbitrary A in **AP-PRIV**, we simplify the definition of the other policies, as they otherwise would need to define special rules for `private`-like access.

Current Instance. Now, we adapt these rules to match the Java semantics. First, Figure 14d shows how to prevent access to the private field on instances of subclasses (Figure 12). It uses a new type of constraint, $p \sim R$, which holds when the sequence of labels in path p is in the language described by the regular expression R . In this case, we assert that the access path p must adhere to the regular expression `LEX` * . This prevents access from instances of subclasses of the defining class, as that requires traversing an `EXT` edge. For example, the access path in Figure 12 would be $s_B \xrightarrow{\text{EXT}} s_A \sim \text{LEX}^*$, which is not satisfiable.

Outermost Class. Finally, Figure 14e shows how to expose `private` fields to the outermost enclosing class. In this rule, the set $S_{C_{ref}}$ contains the scope of the enclosing classes of the reference location, and $S_{C_{decl}}$ contains the scope of the enclosing classes of the class in which the declaration occurs. These sets should share a scope s_c , which represents the shared enclosing class of the reference and the declaration.

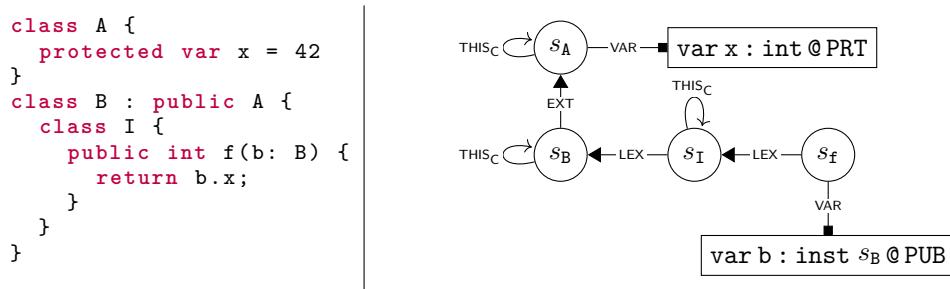
$$\langle acc \rangle ::= \dots \mid \text{protected} \quad \langle A \rangle ::= \dots \mid \text{PRT}$$

(a) Syntax of `protected` keyword.

$$\text{A-PROT} \frac{}{s \vdash_g \text{protected} \Rightarrow \text{PRT}} \quad \text{AP-PROT} \frac{\vdash_g s \uparrow_C S_C \quad s_c \in S_C \quad s_c \in \text{scopes}(p)}{s \vdash_g p ! \text{PRT}}$$

(b) Semantics of `protected` keyword.

■ **Figure 15** Extending AML (Figure 7) with `protected` visibility.



(a) Example program and partial scope graph demonstrating the `protected` access modifier.

$$\frac{\vdash_g s_f \uparrow_C \{ s_I, s_B \} \quad s_B \in \{ s_I, s_B \} \quad s_B \in \text{scopes}(s_B \xrightarrow{\text{EXT}} s_A)}{s_f \vdash_g (s_B \xrightarrow{\text{EXT}} s_A) ! \text{PRT}}$$

(b) Part of typing derivation that shows how access is granted by the PRT accessibility policy.

■ **Figure 16** Example program demonstrating the meaning of the `protected` access modifier.

Note how this rule enables type-checking the program in Figure 13. Using AP-PRIV does not work, as $\vdash_g s_A \uparrow_C \{ s_A \}$, which does not include $\text{tgt}(p) = s_B$. However, we can check it with AP-PRIV", as $\vdash_g \text{tgt}(p) \uparrow_C \{ s_B, s_A \}$, which includes the shared enclosing class s_A .

6.2 Protected Access

The `protected` access modifier (Figure 15a) grants access to subclasses of the defining class, including classes nested in subclasses. For field access expressions ($\langle e.x \rangle$), e must be an instance of a class that encloses the reference [11, §6.6.2.1]. This semantics (Figure 15b) can be modeled by asserting that there should be some class s_c that is both (a) an enclosing scope of the reference location ($\vdash_g s \uparrow_C S_C$), and (b) occurs in the in the access path ($s_c \in \text{scopes}(p)$). The last condition implies that the enclosing class s_c is a subclass of the *defining class*, which is the intuitive understanding of the `protected` keyword.

Figure 16 demonstrates how this rule works. In this program, there is a class A which has a subclass B. Class B has a nested class I, which has a method f with a parameter b of type B. The body of f accesses field x on the instance of B. On the right-hand side of the picture, a part of the corresponding scope graph is shown. The scopes for classes A and B are connected by an EXT-edge again. The fact that class I is nested in class B is represented by the $s_I \xrightarrow{\text{LEX}} s_B$ edge, similar to other lexically nested constructs. Likewise, scope s_f , which represents the body of the method f, has a LEX-edge to s_I .

Figure 16b shows how the access to $b.x$ is validated. The first premise states that s_I and s_B are the enclosing classes of s_f . The other premises assert that s_B is in the enclosing classes as well as in the access path. Together, this allows access to the protected member. Note how access to an instance of A in s_f would not be allowed. In that case, the access path would have been just s_A , which is not an enclosing class of s_f .

47:18 Defining Name Accessibility Using Scope Graphs

$\langle acc \rangle ::= \dots | \text{protected internal} (\langle x \rangle^*) | \text{private protected} (\langle x \rangle^*)$

$\langle A \rangle ::= \dots | \text{SMD } S | \text{SMC } S$

(a) Syntax of policy interaction keywords.

$$\text{A-PPROT} \frac{S = \left\{ s' \mid x_i \in \bar{x}_{0\dots n}, s \vdash_{\mathcal{G}} x_i \xrightarrow{\text{M}} s' \right\}}{s \vdash_{\mathcal{G}} \text{private protected}(\bar{x}_{0\dots n}) \Rightarrow \text{SMC } S}$$

$$\text{A-PINT} \frac{S = \left\{ s' \mid x_i \in \bar{x}_{0\dots n}, s \vdash_{\mathcal{G}} x_i \xrightarrow{\text{M}} s' \right\}}{s \vdash_{\mathcal{G}} \text{protected internal}(\bar{x}_{0\dots n}) \Rightarrow \text{SMD } S}$$

(b) Translation of composite keywords to their policies.

$$\text{AP-SMC} \frac{s \vdash_{\mathcal{G}} p ! \text{MOD } S \quad s \vdash_{\mathcal{G}} p ! \text{PRT}}{s \vdash_{\mathcal{G}} p ! \text{SMC } S}$$

$$\text{AP-SMD-PROT} \frac{s \vdash_{\mathcal{G}} p ! \text{PRT}}{s \vdash_{\mathcal{G}} p ! \text{SMD } S} \quad \text{AP-SMD-MOD} \frac{s \vdash_{\mathcal{G}} p ! \text{MOD } S^{(*)}}{s \vdash_{\mathcal{G}} p ! \text{SMD } S}$$

(c) Semantics of interaction policies.

Figure 17 Extending AML (Figure 7) with keywords to combine module-level and subclass-level accessibility.

7 Combining Subclass and Module Visibility

Access modifiers regulating both the module and subclass dimensions occur in real-world languages as well. For example (as noticed earlier), Java's `protected` keyword also exposes a definition in the same package, similar to C#'s `protected internal`. In addition, C# has a `private protected` modifier, which allows access to subclasses in the same assembly only. In fact, those two keywords denote the two main ways in which access modifiers can interact. First, `protected internal` denotes *disjunctive* interaction, where a declaration is accessible from the subclasses *or* the same module. Second, `private protected` denotes *conjunctive* interaction, where a declaration is accessible from the subclasses *in* the same module only. These interactions are straightforward to define, with one intricate case discussed below.

Figure 17a defines the syntax of the two new keywords (based on their name in C#) and policies. We add `SMD` (Subclass/Module, Disjunctive) and `SMC` (Subclass/Module, Conjunctive) policies, which each contain a list of module scopes to which they are exposed. The translation from keyword to policy is given in Figure 17b. Both rules resolve their module arguments, similar to `A-INT`. The `SMC` policy has one rule (`AP-SMC`), which simply asserts that access is granted by the module (`MOD`) and protected (`PRV`) policies. There are two rules for the `SMD` policy. The first one simply delegates to the `PRT` access policy, permitting access wherever a `protected` member would have been accessible. The other rule delegates to the `MOD` policy, but more careful attention must be paid here (hence the $(*)$ mark). Recall that the semantics of this policy has a variant that asserts that the whole inheritance chain has access to the declaration (Figure 10e). However, this extension should *not* be applied here, because the `protected` part of this modifier already grants access, regardless of the module-level exposure.

$$\begin{array}{c}
 \text{P-PUB} \frac{p \sim \text{LEX}^* \text{EXT}^*}{s \vdash_{\mathcal{G}} p \text{ i}} \quad \text{P-PRIV-PROT} \frac{\vdash_{\mathcal{G}} s \uparrow_C S_C \quad s_c \in S_C \quad \text{split-at}(s_c, p) = \langle p_1, p_2 \rangle}{p_1 \sim \text{LEX}^* \text{EXT}^* \quad p_2 \sim \text{EXT}_{\text{PRV}}^? (\text{EXT} | \text{EXT}_{\text{PRT}})^*} \\
 \end{array}$$

■ **Figure 18** Extending AML (Figure 7) with path-level visibility.

8 Defining Extends-Clause Accessibility Restriction

Until now, we have only considered inheritance as it exists in Java and C#. In this section, we shift our focus to C++, in particular the access modifiers on extends clauses. In C++, it is possible to add a `private` modifier to an extends clause, which reduces the accessibility of `public` and `protected` members to `private` in the derived class. Similarly, the `protected` keyword can be used to reduce the accessibility of `public` members to `protected`. For qualified accesses, C++ imposes the additional constraint that the inheritance chain leading to class in which the variable is declared should be accessible from the class in which the access occurs [8, §11.9.3 (4)].

Setup. In contrast to the previous sections, we cannot encode inheritance-imposed access control in our accessibility policy *A*. Instead, we encode it in the scope graph directly. For that purpose, we introduce two new labels: `EXTPRV` and `EXTPRT`, which model private and protected extension, respectively. Similar to the previous sections, `EXT` will model public extension; i.e. inheritance without access restriction.

Fortunately, we can validate path access independently from the declaration-level access policy.⁷ We require two adaptations to the rules **T-VAR** and **T-FLD** (Figure 7). First, the regular expressions of the queries must be changed to also traverse these new edges. Thus, in **T-VAR**, `LEX*EXT*VAR` must be changed to `LEX*(EXT|EXTPRT|EXTPRV)*VAR`. Similarly, **T-FLD** now has `(EXT|EXTPRT|EXTPRV)*VAR` as regular expression instead of `EXT*VAR`. Second, we add a premise $s \vdash_{\mathcal{G}} p \text{ i}$ to both rules. This premise asserts that the labels in the path p do not hide the accessed definition in scope s .

Path accessibility can be captured in two rules, shown in Figure 18. First, **P-PUB** asserts that a path is valid when there is only public inheritance. With this rule, the semantics of the programs that do not use private or protected inheritance has not changed. Second, rule **P-PRIV-PROT** covers the other two cases. This rule looks intricate, but the intuition behind it is not too complicated. Similar to the `private` and `protected` modifiers (Sections 6.1 and 6.2), access must occur within the class where the member is `private/protected`. This is now not necessarily the defining class, but rather the last class in the inheritance chain that has a non-public modifier on the extends clause. In the rule, this is encoded as follows. The first two premises introduce a scope s_c , which is an enclosing scope of the reference location s . The third premise asserts that the path p can be split into two segments at scope s_c . That is, p consists of two segments: a part p_1 from s_1 to s_c and a part p_2 from s_c to s_n . This implies that s_c is in the access path. To validate that all subclasses of s_c in the path have public inheritance, p_1 should match regular expression `LEX*EXT`.⁸ The path leading from the current class to the declaration (p_2) may start with a private inheritance step (`EXTPRV?`), but may have only public and protected inheritance higher in the access path.

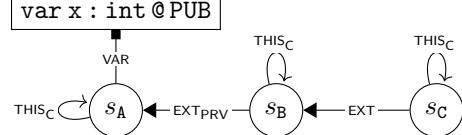
⁷ That also holds for the subtle interaction between `internal` and `protected` discussed in Section 7. `protected` or `private` inheritance in subclasses of the reference class can still compromise these access modes, and must therefore be validated.

⁸ Alternatively, one can encode the requirement that the instance type must be s_c itself by using `LEX*`, similar to Figure 14d.

```

class A {
    public var x = 42
}
class B : private A {
    public var y = new C().x
}
class C : public B { }

```



(a) Example program and partial scope graph demonstrating path access restrictions.

$$\frac{\dots}{\vdash_{\mathcal{G}} s_B \uparrow_{s_C} \{s_B\}} \quad s_B \in \{s_B\}$$

$$\text{split-at}(s_B, s_C \xrightarrow{\text{EXT}} s_B \xrightarrow{\text{EXT}_{\text{PRV}}} s_A) = \langle s_C \xrightarrow{\text{EXT}} s_B, s_B \xrightarrow{\text{EXT}_{\text{PRV}}} s_A \rangle$$

$$\frac{(s_C \xrightarrow{\text{EXT}} s_B) \sim \text{LEX}^* \text{EXT}^* \quad (s_B \xrightarrow{\text{EXT}_{\text{PRV}}} s_A) \sim \text{EXT}_{\text{PRV}}? (\text{EXT} | \text{EXT}_{\text{PRT}})^*}{s_B \vdash_{\mathcal{G}} (s_C \xrightarrow{\text{EXT}} s_B \xrightarrow{\text{EXT}_{\text{PRV}}} s_A) \downarrow}$$

(b) Part of typing derivation that shows how access is granted by the P-PRIV-PROT rule.

Figure 19 Example program demonstrating path accessibility.

Figure 19 gives an example that uses this rule. There is a class A with a field x. Class A is inherited privately by class B, which makes x private in B. Next, class C extends B publicly. In class B, x is accessed on an instance of C. This access should be allowed, as class B is the class in which x is private as well as the class in which the reference occurs. The partial derivation in Figure 19b asserts this. s_B is the scope that encloses the reference. Splitting the access path from s_C to s_A at that s_B yields two segments of a single step. The segment leading up to s_B ($s_C \xrightarrow{\text{EXT}} s_B$) does indeed match the regular expression $\text{LEX}^* \text{EXT}^*$. Likewise, the other segment also matches its regular expressions, showing that this access is valid. Note that, when class C would have extended class B with **protected** or **private** visibility instead, the premise on the first section would not hold anymore. This corresponds with the behavior in Section 6 (the field must be accessible as if it was defined on the instance type) as well as the specification of C++ cited above.

9 Analysis

A comprehensive model of accessibility can be made by composing the system fragments we discussed so far (Figures 7, 10, 14, 15, 17, and 18). In this section, we discuss a few properties that our system adheres to.

9.1 Soundness of Access Policies

First, we claim some soundness theorems for **private**, **protected** and **internal** access. There is no soundness theorem for **public**, as access is allowed unconditionally. Soundness theorems for **private** **protected** and **protected internal** are easily derived from Theorems 2 and 3, and hence omitted. In the theorems, $P_{\mathcal{G}}$ ranges over valid typing derivation for an AML program with scope graph \mathcal{G} , x_r over references, and x_d over declarations. Appendix D [35] defines the predicates used in these theorems, and proves them.

First, soundness for **private** access is stated as follows:

- **Theorem 1** (Soundness of private member access).

$$\begin{aligned} \text{resolve}_{P_G}(x_r) = x_d \wedge \text{private}_{P_G}(x_d) &\Rightarrow \\ \exists s_d. \text{definingClass}_{P_G}(x_d) = s_d \wedge \text{enclosingClass}_{P_G}(x_r, s_d) \end{aligned}$$

This should be read as “when x_r resolves to x_d , and x_d is private, then x_r must occur in the class s_c that defines x_d ”.

Likewise, soundness for **protected** access is stated as:

- **Theorem 2** (Soundness of protected member access).

$$\begin{aligned} \text{resolve}_{P_G}(x_r) = x_d \wedge \text{protected}_{P_G}(x_d) &\Rightarrow \\ \exists s_c, s_d. \text{definingClass}_{P_G}(x_d) = s_d \wedge \text{enclosingClass}_{P_G}(x_r, s_c) \wedge \text{subClass}_{P_G}(s_c, s_d) \end{aligned}$$

Compared to Theorem 1, this theorem states that x_r can occur in some arbitrary subclass s_c of s_d if x_d is **protected**.

Finally, **internal** access is specified correctly when:

- **Theorem 3** (Soundness of internal member access).

$$\begin{aligned} \text{resolve}_{P_G}(x_r) = x_d \wedge \text{internal}_{P_G}(x_d, \bar{x}) &\Rightarrow \\ (\exists x, s_m. \text{enclosingMod}_{P_G}(x_r) = s_m \wedge x \in \bar{x} \wedge \text{resolveMod}(x) = s_m) \vee \\ (\exists s_d. \text{definingClass}_{P_G}(x_d) = s_d \wedge \text{enclosingClass}_{P_G}(x_r, s_d)) \end{aligned}$$

This theorem states that references to declarations with modifier **internal** are valid if the enclosing module of the reference s_m is referred to in the arguments of the access modifier \bar{x} , or if it is accessed as a private variable.

9.2 Equivalence of Access Policies

The access policy language $\langle A \rangle$ we defined is not minimal. It is possible to define equivalent policies in multiple ways. To analyze that, we define equivalence of access policies as follows:

- **Definition 4** (Equivalence of Access Policies).

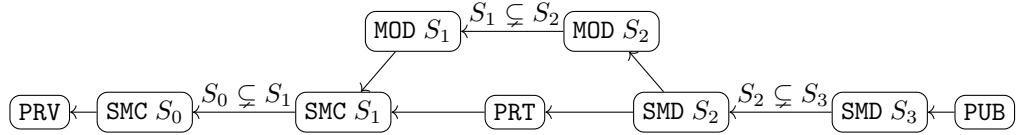
$$\frac{\forall \mathcal{G}, s, p. (s \vdash_{\mathcal{G}} p ! A) \Leftrightarrow (s \vdash_{\mathcal{G}} p ! A')}{A \equiv A'}$$

That is, two accessibility policies are equivalent when, for any scope s , path p , scope graph \mathcal{G} , either both policies admit access, or neither does.

The equivalences that hold in our model are: **PRT** \equiv **SMD** \emptyset and **PRV** \equiv **SMC** \emptyset \equiv **MOD** \emptyset . This follows from the fact that module access grants nothing if no module parameters are given. Thus, the **SMD** \emptyset policy reduces to **PRT**, while **SMC** \emptyset and **MOD** \emptyset do not elevate accessibility beyond **PRV**. Appendix B [35] gives proofs for each of these equivalences. Because of these equivalences, we did not include **PRT** and **PRV** in our implementation (Section 10).

9.3 Order of Access Policies

Intuitively, there exists an ordering between accessibility policies, where **PRV** is the bottom most restrictive, and **PUB** is the least restrictive. This order is partial, as the module-exposure dimension and subclass-exposure dimension are orthogonal. Assuming a subset relation on scope sets ($S \subset S'$), we can define a strict partial order $A <_A A'$ as follows:



where the edges indicate instances of the $<_A$ -relation. The edges with a condition indicate that SMC, MOD, and SMD become more permissive when more scopes are added to the policy.

The intuition behind this order is not arbitrary. In fact, we claim the following:

► **Theorem 5** (The order on access policies $<_A$ is well-behaved).

$$(A <_A A') \Rightarrow \forall \mathcal{G}, s, p. (s \vdash_{\mathcal{G}} p ! A) \Rightarrow (s \vdash_{\mathcal{G}} p ! A')$$

That is, when A is more restrictive than A' , and A permits access in scope s via a path p , then A' will permit that access too. A proof of this theorem can be found in Appendix C [35].

10 Evaluation

So far, we have motivated our specification with examples from real-world languages such as Java and C#, and stated some generic properties of our model. However, for our specification to be usable as a basis for practical tools, it must correspond with the behavior of the actual languages. To validate that, we evaluated our specification in two ways. First, we systematically compared our specification with reference compilers of Java, C#, and Rust. Second, we validated the compatibility of our framework with recent work on language-parametric code completion [19].

10.1 Comparison with Reference Compilers: Implementation

The comparison to compilers of real-world languages is implemented as follows:

1. Apply our type system on an AML program (the test case).
2. Translate the AML program to the target language.
3. Compile the translated program using a compiler of the target language.
4. Compare results: either both analyses should succeed, or both should give errors.

We discuss these steps in more detail below.

AML Type Checker. To compare our model with real-world compilers, we need a way to type check concrete AML programs. To that end, we implemented AML in the Spooftax language workbench [13, 31]. The actual type system is implemented using the Statix specification language [27, 22]. Statix is a suitable choice, as its declarativity allows an overall straightforward translation from our inference. For example, the Statix encoding of rule **T-VAR** in Figure 20a strongly corresponds to the original (Figure 7). Using this implementation, we can systematically check accessibility in concrete AML programs.

Compiling with Reference Compiler. Next, we implemented source-to-source translations from AML to each of Java, C# and Rust. This translation was straightforward by design, as otherwise the results of the type checkers can be different due to semantic differences introduced by the translation. For that reason, we do not support AML features that have no direct counterpart in the target language. For example, the translation to Java will error when the AML program uses the `private protected` access modifier, as Java does not support that accessibility policy. This way, we know that correspondence between the programs is guaranteed when the translation succeeds.

```

typeOfExpr: scope * Expr -> TYPE
typeOfExpr(s, Id(x)) = T :- fp A}
query var
filter LEX* EXT*
and { x' :- x' == x }
min $ < EXT, EXT < LEX
in s |-> [(p, (x, T, A))],
accessOk(s, p, A),
pathOk(s, p).

```

(a) Encoding of rule **T-VAR** in Statix.

```

test private - nested [[
class A {
private var x = 42
class B {
public var y = x
}
]]
analysis succeeds
run java-compat

```

(b) Example test case.

Figure 20 Overview of Approach to Comparison with Reference Compilers.

After translating, we invoke the reference compiler, observe its output (success or failure), and compare the given output with the result from our own type checker (step 1). If those are different (i.e., our type checker accepts the program, while the reference compiler emits errors, or vice versa), the test fails.

10.2 Comparison with Reference Compilers: Test Cases

To the best of our knowledge, there exists no test suite specifically aimed at verifying the semantics of access modifiers. For that reason, we manually created an extensive test suite. Each test contains a class **Def**, that defines some variable **x** with some access modifier **A**. Furthermore, each test contains a class **Ref**, in which a reference to **x** occurs. **Def** and **Ref** can be related in two different ways at the same time:

- By inheritance: either (1) **Def** and **Ref** are actually the same class, (2) have no mutual inheritance, (3) **Ref** inherits **Def**, or (4) **Def** inherits **Ref**.
- By module position: either **Def** and **Ref** (1) occur in the same module, or (2) **Ref** occurs in a parent/sibling/child module of **Def**.
- By class nesting: either (1) **Def** and **Ref** are top-level classes, (2) **Ref** is nested in **Def**, (3) **Def** is nested in **Ref**, or (4) **Def** and **Ref** have a shared enclosing class.

In addition, tests for member accesses (i.e., **recv.x**) have a receiver type **Recv**. This type must either be equal to **Def**, or inherit from it. However, it can be related in all possible other ways to **Def** and **Ref**. By systematically exploring all options, we derived our test suite.

We excluded cases that are (1) impossible (e.g., **Ref** cannot be nested in **Def** and live in a different module at the same time), (2) use features not supported by the target language, (3) invalid for another reason (i.e., inheriting from a nested class is not allowed by Java), or (4) do not bind properly (i.e., lexical access where **Ref** and **Def** do not inherit from each other, and are not nested in each other). To reduce the number of test cases, we restricted the cases that involved nested classes to have one module only. Additionally, we only used **private**, **protected** and **protected internal** as access modifiers in these cases. Table 2 summarizes the results of the test suite generation.

Figure 20b shows an example test case written in the Spooftax Testing Language (SPT) [12]. This test validates that a private field is accessible from a nested class. The test consists of a program (between double square brackets), and some *expectations*. In this case, we expect the (Statix-based) analysis to succeed. Moreover, we expect the **java-compat** transformation to succeed. This transformation is executing the steps in Section 10.1.

 **Table 2** Summary of Test Suite.

	Java	C#	Rust	Manual
<i>Acc. Mods.</i>	<code>public</code>	Same as Java, and	<code>public</code>	All
	<code>protected internal</code>	<code>protected</code>	<code>internal</code>	
	<code>internal, private</code>	<code>private protected</code>		
<i>Features</i>	class inheritance	class inheritance	structs, modules	advanced modules
	class nesting, and packages	class nesting, and assemblies		inheritance visibility
<i>#Cases</i>	433	522	60	168
<i>Compiler</i>	<code>javac 11.0.20.1</code>	<code>dotnet 7.0.401</code>	<code>rustc 1.73.0</code>	—

Results. There are several features present in AML that were not covered by any of the reference compilers, most notably *private/protected inheritance*, and module visibility beyond what Rust supports. To validate we cover these features to some extent, we have written 168 additional test cases. While initially exposing a lot of edge cases, in the end all test cases succeeded. This shows that our specification covers the languages it set out to model rather accurately.

10.3 Code Completion

One of our future goals is to use our framework to implement refactoring tools that are sound with respect to accessibility. The most recent work in this area is done by Pelsmaeker et al. [19]. They show how Statix specifications can be used to generate editor auto-completion proposals language-parametrically. We applied auto-completion to the access modifiers in the C#/Java and Rust tests (152, after deduplication), and validated soundness and completeness. That is, when the analysis succeeded, code completion should propose the current modifier at that position. Otherwise, if the access was invalid, the modifier should not be proposed, as only less restrictive ones are valid at that position.

We consider the fact that all completion tests pass a good indication that our specification can be applied with refactoring tools in the future. Apparently, the code completion framework is sound and complete with respect to our encoding of access modifiers. Accessibility errors introduced by a refactoring can be fixed by generating proposals for that position, and using the ordering from Section 9.3 to pick the most restrictive one.

10.4 Threats to Validity

In Section 4, we briefly mentioned that the specification as presented in the paper did not model the interaction between shadowing and accessibility correctly. Doing so would require a full path order, instead of ordering paths by a lexicographical order on labels. Appendix A.1 [35] explains how we think that could be done. However, Statix does not support full path orders. To work around that, we emulated this behavior using a few helper predicates. Our test suite gives confidence we modeled it correctly, but we did not prove that the specification in Statix and the full path order are semantically equivalent.

Finally, we might have modeled incorrect/unspecified behavior if the reference compilers were incorrect. Examples such as Figure 12 were derived from actual compiler behavior. However, we could not find our interpretation of the implementation behavior explicitly specified in the JLS [11, §6.6.1].

11 Related Work

In this section, we discuss previous work related to access modifiers and scope graphs.

11.1 Access Modifier Semantics and Implementations

The origin of access modifiers dates back to at least Simula 67, which around 1972 introduced `protected` and `hidden` access modifiers [4, §8] (the latter being equivalent to our `private`). Later, languages such as Java and C++ incorporated these keywords, making them well-known and often used. Design principles and patterns [9] using these keywords were developed, making contemporary software development heavily reliant on accessibility features the programming language provides.

Giurca and Savulea (2004) [10] apply object-oriented notions of `public`, `protected` and `private` to logic programs, with the purpose of better knowledge distribution and run time optimization. Moreover, Apel et al. [2, 1] introduce access modifiers in feature-oriented programming. Where we define accessibility for module nesting and class inheritance, they add the “feature refinement” dimension to this. In particular, the `feature` keyword restricts access to the “current feature” only (comparable to `private` in the class inheritance dimension), the `subsequent` keyword grants access to the current feature and later refinements (comparable to `protected`), and the `program` modifier allows access from any position (similar to `public`). In our terminology, their model supports “conjunctive” combination of the class and feature dimensions. As Section 7 shows that combining the module and class dimensions conjunctively is straightforward, we expect that integrating their work in our model will not pose major challenges (apart from a combinatorial explosion of policies).

Semantics. As access modifiers mainly originated from practical needs, it is not very surprising that little attention to them was paid from a more theoretical perspective. A few attempts to create a more formal account have been performed, however. In 1998, Yang [33] presented a formalization of Java access modifiers using attribute grammars. At that time, attribute grammars still lacked several convenience features, such as default attributes [30] and collection attributes [14]. For that reason, all members must be propagated explicitly to the scopes where they are accessible, which makes the specification rather verbose. Additionally, since fields and methods are not treated equally (shadowing vs. overriding), they are treated separately, doubling the specification size. In contrast, we specify the propagation of members queries in scope graphs, which is more concise. The additional requirements on methods (not explicitly discussed), can be handled at the definition site. Furthermore, we cover more features than just the Java ones. Pharkani et al. [6] present a generalized model of accessibility, where accessibility is modeled as a set of rules granting access of a member to another member (similar to Eiffel/`friends` in C++). In addition, rules can *deny* access to the named member, or apply to all members except the named ones.

Tools. Steimann et al. [25] observe that disregarding accessibility can result in a lot of subtle mistakes. For example, a method may silently fail to override another method when it is moved to a different package, which results in different dynamic dispatch. To capture these errors, they present nine constraint generation rules that model the accessibility semantics of Java. Refactoring tools can use these constraints to detect where the accessibility level of a member must be elevated. This work was incorporated in the JRRT refactoring tool [23], which was evaluated on a large number of real-world Java projects, showing the accuracy of their implementation. While their work also covers overriding-specific constraints, which our

specification treats rather superficially, we think our model is more comprehensible, and also gives insight in the differences between languages. Moreover, their work is applied in real refactoring implementations, while the quest for Statix-based refactorings is still ongoing. Meanwhile, a similar approach was applied to Eiffel accessibility [24, §6.3].

While these tools *elevate* accessibility if needed, a different line of research aims to *restrict* accessibility if possible [5, 17, 34]. The purpose of these tools is to detect access modifiers that are more lenient than needed, and restrict those. This is claimed to improve readability, enable optimizations, and increase modularity [17]. The exact underlying model is not the topic of these publications, and hence remains unclear. Despite that, the tools appear to be useful in practice. Zoller and Schmolitzky mention some challenges in porting their tool to other object-oriented languages [34, V.B]. A language-parametric model such as ours helps in that regard by (1) making differences between languages explicit, and (2) make implementations of these (kind of) tools language-parametric.

11.2 Scope Graphs

Scope Graphs (Section 3) have been introduced by Neron et al. [18], and later refined by Van Antwerpen et al. [27] and Rouvoet et al. [22]. In order to bridge the gap between language specification and implementation, scope graphs have been embedded into the NaBL2 constraint language [26]. Later, the Statix logic language was introduced [27, 22], which is more expressive than NaBL2. Both languages allow specifying type checking as constraint programs, giving the language a declarative appeal, but also yielding an executable type checker. Scope graphs are also available in a framework for concurrent and incremental type checkers [28, 39] and an embedded DSL in Haskell [20]. Finally, Statix specifications have been used for language-parametric code completion [19] and refactorings [16, 29]. Zwaan and Van Antwerpen provide a detailed overview of the development history of scope graphs, their embeddings in type system specification DSLs, and their applications [38].

12 Conclusion

Access modifiers occur in many real-world languages. To implement high-quality tooling for these languages, a good understanding of access modifiers is required. In this paper, we presented a model for access validation based on scope graphs. Our model covers the most important accessibility features in contemporary languages, including module accessibility, and inheritance accessibility, both on declarations and extends-clauses. Variations between different languages, both in supported features and their semantics, are made explicit in our model. Our specification is quite declarative, partly because scope graphs abstract over low-level name resolution and scoping details. Our model was validated using an extensive test suite, using Java, C#, and Rust compilers as oracles. This test suite was also used to show that we can synthesize access modifiers accurately using previous work on code completion [19].

Our main motivation for this work is twofold. First, we aim to provide a “language-transcendent” model for accessibility that enables comparison of different languages regarding accessibility. To this end we identify and formalize differences in the semantics of several access modifiers. In addition, we formulate soundness theorems of several access modifiers, and prove them. As such, we consider our specification accurate enough to serve as a reference for future tool implementations. Second, we aim to use our model in language-parametric refactorings, ensuring they respect accessibility properly. As these refactoring tools are still in development, actual validation of this application is still future work.

References

- 1 Sven Apel, Sergiy S. Kolesnikov, Jörg Liebig, Christian Kästner, Martin Kuhlemann, and Thomas Leich. Access control in feature-oriented programming. *Science of Computer Programming*, 77(3):174–187, 2012. doi:10.1016/j.scico.2010.07.005.
- 2 Sven Apel, Jörg Liebig, Christian Kästner, Martin Kuhlemann, and Thomas Leich. An orthogonal access modifier model for feature-oriented programming. In Sven Apel, William R. Cook, Krzysztof Czarnecki, Christian Kästner, Neil Loughran, and Oscar Nierstrasz, editors, *Proceedings of the First International Workshop on Feature-Oriented Software Development, FOSD 2009, Denver, Colorado, USA, October 6, 2009*, ACM International Conference Proceeding Series, pages 27–33. ACM, 2009. doi:10.1145/1629716.1629723.
- 3 Casper Bach Poulsen, Xulei Liu, and Luka Miljak. Towards a Language-parametric DSL for Refactoring (Short Paper), 2024. URL: https://pop124.sigplan.org/details?action-call-with-get-request-type=1&c9432bfaa61a48fb852237f9e99a821daction_1742650661080820307cb713fc2d28c30ae360b0bed=1&__ajax_runtime_request_=1&context=POPL-2024&track=pepm-2024&urlKey=8&decoTitle=Towards-a-Language-parametric-DSL-for-Refactoring-Short-Paper-.
- 4 Andrew P. Black. Object-oriented programming: Some history, and challenges for the next fifty years. *Inf. Comput.*, 231:3–20, 2013. doi:10.1016/j.ic.2013.08.002.
- 5 Philipp Bouillon, Eric Grobkinsky, and Friedrich Steimann. Controlling Accessibility in Agile Projects with the Access Modifier Modifier. In Richard F. Paige and Bertrand Meyer, editors, *Objects, Components, Models and Patterns, 46th International Conference, TOOLS EUROPE 2008*, volume 11 of *Lecture Notes in Business Information Processing*, pages 41–59. Springer, 2008. doi:10.1007/978-3-540-69824-1_4.
- 6 Toktam Ramezani Farkhani, Mohammadreza Razzazi, and Peyman Teymoori. Eam: Expansive access modifiers in oop. In *2008 International Conference on Computer and Communication Engineering*, pages 589–594, 2008. doi:10.1109/ICCCE.2008.4580672.
- 7 The Rust Foundation. The Rust Reference. Accessed 25-09-2023. URL: <https://doc.rust-lang.org/reference/>.
- 8 The Standard C++ Foundation. Working Draft, Standard for Programming Language C++. Online version from <https://github.com/cplusplus/draft/releases/tag/n4868> was consulted. Per release notes, ‘only editorial changes compared to C++20’ were made.
- 9 Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In Oscar M. Nierstrasz, editor, *ECOOP’93 — Object-Oriented Programming*, pages 406–431, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg. doi:10.1007/3-540-47910-4_21.
- 10 Adrian Giurca and Dorel Savulea. Logic programs with access modifiers. In *4th International Conference on Artificial Intelligence and Digital Communication, AIDC*, pages 22–31, 2004.
- 11 James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The Java Language Specification - Java SE 8 Edition, February 2015. URL: <https://docs.oracle.com/javase/specs/jls/se8/html/>.
- 12 Lennart C. L. Kats, Rob Vermaas, and Eelco Visser. Integrated language definition testing: enabling test-driven language development. 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 139–154. ACM, 2011. doi:10.1145/2048066.2048080.
- 13 Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. 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*, pages 444–463, Reno/Tahoe, Nevada, 2010. ACM. doi:10.1145/1869459.1869497.

47:28 Defining Name Accessibility Using Scope Graphs

- 14 Eva Magnusson, Torbjorn Ekman, and Goren Hedin. Extending Attribute Grammars with Collection Attributes—Evaluation and Applications. *Source Code Analysis and Manipulation, IEEE International Workshop on*, 0, 2007. doi:10.1109/SCAM.2007.13.
- 15 Luka Miljak, Casper Bach Poulsen, and Flip van Spaendonck. Verifying Well-Typedness Preservation of Refactorings using Scope Graphs. In Aaron Tomb, editor, *Proceedings of the 25th ACM International Workshop on Formal Techniques for Java-like Programs, FTfJP 2023, Seattle, WA, USA, 18 July 2023*, pages 44–50. ACM, 2023. doi:10.1145/3605156.3606455.
- 16 Phil Misteli. Renaming for Everyone: Language-parametric Renaming in Spoofax. Master’s thesis, Delft University of Technology, May 2021. URL: <http://resolver.tudelft.nl/uuid:60f5710d-445d-4583-957c-79d6afa45be5>.
- 17 Andreas Müller. Bytecode analysis for checking java access modifiers. In *Work in Progress and Poster Session, 8th Int. Conf. on Principles and Practice of Programming in Java (PPPJ 2010), Vienna, Austria*, pages 1–4, 2010.
- 18 Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A Theory of Name Resolution. In Jan Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 205–231. Springer, 2015. doi:10.1007/978-3-662-46669-8_9.
- 19 Daniël A. A. Pelsmaeker, Hendrik van Antwerpen, Casper Bach Poulsen, and Eelco Visser. Language-parametric static semantic code completion. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA):1–30, 2022. doi:10.1145/3527329.
- 20 Casper Bach Poulsen, Aron Zwaan, and Paul Hübner. A Monadic Framework for Name Resolution in Multi-phased Type Checkers. In Coen De Roover, Bernhard Rümpe, and Amir Shaikhha, editors, *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2023, Cascais, Portugal, October 22-23, 2023*, pages 14–28. ACM, 2023. doi:10.1145/3624007.3624051.
- 21 Mark Reinhold. Java platform module system, August 2017. URL: <https://jcp.org/en/jsr/detail?id=376>.
- 22 Arjen Rouvoet, Hendrik van Antwerpen, Casper Bach Poulsen, Robbert Krebbers, and Eelco Visser. Knowing when to ask: sound scheduling of name resolution in type checkers derived from declarative specifications. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), 2020. doi:10.1145/3428248.
- 23 Max Schäfer, Andreas Thies, Friedrich Steimann, and Frank Tip. A Comprehensive Approach to Naming and Accessibility in Refactoring Java Programs. *IEEE Trans. Software Eng.*, 38(6):1233–1257, 2012. doi:10.1109/TSE.2012.13.
- 24 Friedrich Steimann, Christian Kollee, and Jens von Pilgrim. A Refactoring Constraint Language and Its Application to Eiffel. In Mira Mezini, editor, *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*, volume 6813 of *Lecture Notes in Computer Science*, pages 255–280. Springer, 2011. doi:10.1007/978-3-642-22655-7_13.
- 25 Friedrich Steimann and Andreas Thies. From Public to Private to Absent: Refactoring Java Programs under Constrained Accessibility. In Sophia Drossopoulou, editor, *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, volume 5653 of *Lecture Notes in Computer Science*, pages 419–443. Springer, 2009. doi:10.1007/978-3-642-03013-0_19.
- 26 Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In Martin Erwig and Tiark Rompf, editors, *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 49–60. ACM, 2016. doi:10.1145/2847538.2847543.

- 27 Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. Scopes as types. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), 2018. doi:10.1145/3276484.
- 28 Hendrik van Antwerpen and Eelco Visser. Scope States: Guarding Safety of Name Resolution in Parallel Type Checkers. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPICS*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ECOOP.2021.1.
- 29 Loek Van der Gugten. Function Inlining as a Language Parametric Refactoring. Master's thesis, Delft University of Technology, June 2022. URL: <http://resolver.tudelft.nl/uuid:15057a42-f049-4321-b9ee-f62e7f1fda9f>.
- 30 Eric Van Wyk, Oege de Moor, Kevin Backhouse, and Paul Kwiatkowski. Forwarding in Attribute Grammars for Modular Language Design. In R. Nigel Horspool, editor, *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 128–142. Springer, 2002. doi:10.1007/3-540-45937-5_11.
- 31 Guido Wachsmuth, Gabriël Konat, and Eelco Visser. Language Design with the Spoofax Language Workbench. *IEEE Software*, 31(5):35–43, 2014. doi:10.1109/MS.2014.100.
- 32 Bill Wagner, Manuel Zelenka, and Youssef Victor. C# Reference — Keywords — file, November 2022. URL: <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/file>.
- 33 Wuu Yang. Discovering anomalies in access modifiers in java with a formal specification, 1998. URL: <http://dspace.fcu.edu.tw/bitstream/2377/2120/1/ce07ics001998000164.pdf>.
- 34 Christian Zoller and Axel Schmolitzky. Measuring Inappropriate Generosity with Access Modifiers in Java Systems. In *2012 Joint Conference of the 22nd International Workshop on Software Measurement and the 2012 Seventh International Conference on Software Process and Product Measurement, Assisi, Italy, October 17-19, 2012*, pages 43–52. IEEE Computer Society, 2012. doi:10.1109/IWSM-MENSURA.2012.15.
- 35 Aron Zwaan and Casper Bach Poulsen. Defining Name Accessibility using Scope Graphs (Artifact), May 2024. doi:10.5281/zenodo.11179594.
- 36 Aron Zwaan and Casper Bach Poulsen. Defining Name Accessibility using Scope Graphs (Extended Edition). *CoRR*, May 2024. doi:10.48550/arXiv.2407.09320.
- 37 Aron Zwaan and Casper Bach Poulsen. Defining Name Accessibility using Scope Graphs (Artifact). Software (visited on 2024-08-05). URL: <https://zenodo.org/records/11179594>.
- 38 Aron Zwaan and Hendrik van Antwerpen. Scope Graphs: The Story so Far. In Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann, editors, *Eelco Visser Commemorative Symposium, EVCS 2023, April 5, 2023, Delft, The Netherlands*, volume 109 of *OASIcs*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/OASIcs.EVCS.2023.32.
- 39 Aron Zwaan, Hendrik van Antwerpen, and Eelco Visser. Incremental type-checking for free: using scope graphs to derive incremental type-checkers. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):424–448, 2022. doi:10.1145/3563303.

